



Kubernetes Advanced Practical

Kubernetes 进阶实战

马永亮 著

马哥教育CEO马哥（马永亮）撰写，实用性和专业性毋庸置疑

涵盖Kubernetes架构、部署、核心组件、扩缩容、存储与网络策略、安全、系统扩展等话题

Kubernetes主流知识点全覆盖，渐进式讲解，手把手示范，大量实操案例，随时动手验证



机械工业出版社
China Machine Press

云计算与虚拟化技术丛书

Kubernetes进阶实战

马永亮 著

ISBN: 978-7-111-61445-6

本书纸版由机械工业出版社于2019年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

前言

第1章 Kubernetes系统基础

1.1 容器技术概述

- 1.1.1 容器技术的功用
- 1.1.2 容器简史
- 1.1.3 Docker的功能限制

1.2 Kubernetes概述

- 1.2.1 Kubernetes简史
- 1.2.2 Kubernetes特性
- 1.2.3 Kubernetes概念和术语

1.3 Kubernetes集群组件

- 1.3.1 Master组件
- 1.3.2 Node组件
- 1.3.3 核心附件

1.4 Kubernetes网络模型基础

- 1.4.1 网络模型概述
- 1.4.2 集群上的网络通信

1.5 本章小结

第2章 Kubernetes快速入门

2.1 Kubernetes的核心对象

- 2.1.1 Pod资源对象
- 2.1.2 Controller
- 2.1.3 Service
- 2.1.4 部署应用程序的主体过程

2.2 部署Kubernetes集群

- 2.2.1 kubeadm部署工具
- 2.2.2 集群运行模式
- 2.2.3 准备用于实践操作的集群环境
- 2.2.4 获取集群环境相关的信息

2.3 kubectl使用基础与示例

2.4 命令式容器应用编排

- 2.4.1 部署应用（Pod）
- 2.4.2 探查Pod及应用详情
- 2.4.3 部署Service对象
- 2.4.4 扩容和缩容

- 2.4.5 修改及删除对象
- 2.5 本章小结
- 第3章 资源管理基础
 - 3.1 资源对象及API群组
 - 3.1.1 Kubernetes的资源对象
 - 3.1.2 资源及其在API中的组织形式
 - 3.1.3 访问Kubernetes REST API
 - 3.2 对象类资源格式
 - 3.2.1 资源配置清单
 - 3.2.2 metadata嵌套字段
 - 3.2.3 spec和status字段
 - 3.2.4 资源配置清单格式文档
 - 3.2.5 资源对象管理方式
 - 3.3 kubectl命令与资源管理
 - 3.3.1 资源管理操作概述
 - 3.3.2 kubectl的基本用法
 - 3.4 管理名称空间资源
 - 3.4.1 查看名称空间及其资源对象
 - 3.4.2 管理Namespace资源
 - 3.5 Pod资源的基础管理操作
 - 3.5.1 陈述式对象配置管理方式
 - 3.5.2 声明式对象配置管理方式
 - 3.6 本章小结
- 第4章 管理Pod资源对象
 - 4.1 容器与Pod资源对象
 - 4.2 管理Pod对象的容器
 - 4.2.1 镜像及其获取策略
 - 4.2.2 暴露端口
 - 4.2.3 自定义运行的容器化应用
 - 4.2.4 环境变量
 - 4.2.5 共享节点的网络名称空间
 - 4.2.6 设置Pod对象的安全上下文
 - 4.3 标签与标签选择器
 - 4.3.1 标签概述
 - 4.3.2 管理资源标签
 - 4.3.3 标签选择器
 - 4.3.4 Pod节点选择器nodeSelector

- 4.4 资源注解
 - 4.4.1 查看资源注解
 - 4.4.2 管理资源注解
 - 4.5 Pod对象的生命周期
 - 4.5.1 Pod的相位
 - 4.5.2 Pod的创建过程
 - 4.5.3 Pod生命周期中的重要行为
 - 4.5.4 容器的重启策略
 - 4.5.5 Pod的终止过程
 - 4.6 Pod存活性探测
 - 4.6.1 设置exec探针
 - 4.6.2 设置HTTP探针
 - 4.6.3 设置TCP探针
 - 4.6.4 存活性探测行为属性
 - 4.7 Pod就绪性探测
 - 4.8 资源需求及资源限制
 - 4.8.1 资源需求
 - 4.8.2 资源限制
 - 4.8.3 容器的可见资源
 - 4.8.4 Pod的服务质量类别
 - 4.9 本章小结
- 第5章 Pod控制器
- 5.1 关于Pod控制器
 - 5.1.1 Pod控制器概述
 - 5.1.2 控制器与Pod对象
 - 5.1.3 Pod模板资源
 - 5.2 ReplicaSet控制器
 - 5.2.1 ReplicaSet概述
 - 5.2.2 创建ReplicaSet
 - 5.2.3 ReplicaSet管控下的Pod对象
 - 5.2.4 更新ReplicaSet控制器
 - 5.2.5 删除ReplicaSet控制器资源
 - 5.3 Deployment控制器
 - 5.3.1 创建Deployment
 - 5.3.2 更新策略
 - 5.3.3 升级Deployment
 - 5.3.4 金丝雀发布

- 5.3.5 回滚Deployment控制器下的应用发布
 - 5.3.6 扩容和缩容
 - 5.4 DaemonSet控制器
 - 5.4.1 创建DaemonSet资源对象
 - 5.4.2 更新DaemonSet对象
 - 5.5 Job控制器
 - 5.5.1 创建Job对象
 - 5.5.2 并行式Job
 - 5.5.3 Job扩容
 - 5.5.4 删除Job
 - 5.6 CronJob控制器
 - 5.6.1 创建CronJob对象
 - 5.6.2 CronJob的控制机制
 - 5.7 ReplicationController
 - 5.8 Pod中断预算
 - 5.9 本章小结
- 第6章 Service和Ingress
- 6.1 Service资源及其实现模型
 - 6.1.1 Service资源概述
 - 6.1.2 虚拟IP和服务代理
 - 6.2 Service资源的基础应用
 - 6.2.1 创建Service资源
 - 6.2.2 向Service对象请求服务
 - 6.2.3 Service会话粘性
 - 6.3 服务发现
 - 6.3.1 服务发现概述
 - 6.3.2 服务发现方式：环境变量
 - 6.3.3 ClusterDNS和服务发现
 - 6.3.4 服务发现方式：DNS
 - 6.4 服务暴露
 - 6.4.1 Service类型
 - 6.4.2 NodePort类型的Service资源
 - 6.4.3 LoadBalancer类型的Service资源
 - 6.4.4 ExternalName Service
 - 6.5 Headless类型的Service资源
 - 6.5.1 创建Headless Service资源
 - 6.5.2 Pod资源发现

- 6.6 Ingress资源
 - 6.6.1 Ingress和Ingress Controller
 - 6.6.2 创建Ingress资源
 - 6.6.3 Ingress资源类型
 - 6.6.4 部署Ingress控制器 (Nginx)
 - 6.7 案例：使用Ingress发布tomcat
 - 6.7.1 准备名称空间
 - 6.7.2 部署tomcat实例
 - 6.7.3 创建Service资源
 - 6.7.4 创建Ingress资源
 - 6.7.5 配置TLS Ingress资源
 - 6.8 本章小结
- 第7章 存储卷与数据持久化
- 7.1 存储卷概述
 - 7.1.1 Kubernetes支持的存储卷类型
 - 7.1.2 存储卷的使用方式
 - 7.2 临时存储卷
 - 7.2.1 emptyDir存储卷
 - 7.2.2 gitRepo存储卷
 - 7.3 节点存储卷hostPath
 - 7.4 网络存储卷
 - 7.4.1 NFS存储卷
 - 7.4.2 RBD存储卷
 - 7.4.3 GlusterFS存储卷
 - 7.4.4 Cinder存储卷
 - 7.5 持久存储卷
 - 7.5.1 创建PV
 - 7.5.2 创建PVC
 - 7.5.3 在Pod中使用PVC
 - 7.5.4 存储类
 - 7.5.5 PV和PVC的生命周期
 - 7.6 downwardAPI存储卷
 - 7.6.1 环境变量式元数据注入
 - 7.6.2 存储卷式元数据注入
 - 7.7 本章小结
- 第8章 配置容器应用：ConfigMap和Secret
- 8.1 容器化应用配置方式

- 8.2 通过命令行参数配置容器应用
- 8.3 利用环境变量配置容器应用
- 8.4 应用程序配置管理及ConfigMap资源
 - 8.4.1 创建ConfigMap对象
 - 8.4.2 向Pod环境变量传递ConfigMap对象键值数据
 - 8.4.3 ConfigMap存储卷
 - 8.4.4 容器应用重载新配置
 - 8.4.5 使用ConfigMap资源的注意事项
- 8.5 Secret资源
 - 8.5.1 Secret概述
 - 8.5.2 创建Secret资源
 - 8.5.3 Secret存储卷
 - 8.5.4 imagePullSecret资源对象
- 8.6 本章小结
- 第9章 StatefulSet控制器
 - 9.1 StatefulSet概述
 - 9.1.1 Stateful应用和Stateless应用
 - 9.1.2 StatefulSet控制器概述
 - 9.1.3 StatefulSet的特性
 - 9.2 StatefulSet基础应用
 - 9.2.1 创建StatefulSet对象
 - 9.2.2 Pod资源标识符及存储卷
 - 9.3 StatefulSet资源扩缩容
 - 9.4 StatefulSet资源升级
 - 9.4.1 滚动更新
 - 9.4.2 暂存更新操作
 - 9.4.3 金丝雀部署
 - 9.4.4 分段更新
 - 9.4.5 其他话题
 - 9.5 案例: etcd集群
 - 9.5.1 创建Service资源
 - 9.5.2 etcd StatefulSet
 - 9.6 本章小结
- 第10章 认证、授权与准入控制
 - 10.1 访问控制概述
 - 10.1.1 用户账户与用户组
 - 10.1.2 认证、授权与准入控制基础

- 10.2 服务账户管理与应用
 - 10.2.1 Service Account自动化
 - 10.2.2 创建服务账户
 - 10.2.3 调用imagePullSecret资源对象
- 10.3 X.509数字证书认证
 - 10.3.1 Kubernetes中的SSL/TLS认证
 - 10.3.2 客户端配置文件kubeconfig
 - 10.3.3 TLS bootstrapping机制
- 10.4 基于角色的访问控制: RBAC
 - 10.4.1 RBAC授权插件
 - 10.4.2 Role和RoleBinding
 - 10.4.3 ClusterRole和ClusterRoleBinding
 - 10.4.4 聚合型ClusterRole
 - 10.4.5 面向用户的内建ClusterRole
 - 10.4.6 其他的内建ClusterRole和ClusterRoleBinding
- 10.5 Kubernetes Dashboard
 - 10.5.1 部署HTTPS通信的Dashboard
 - 10.5.2 配置token认证
 - 10.5.3 配置kubeconfig认证
- 10.6 准入控制器与应用示例
 - 10.6.1 LimitRange资源与LimitRanger准入控制器
 - 10.6.2 ResourceQuota资源与准入控制器
 - 10.6.3 PodSecurityPolicy
- 10.7 本章小结
- 第11章 网络模型与网络策略
 - 11.1 Kubernetes网络模型及CNI插件
 - 11.1.1 Docker容器的网络模型
 - 11.1.2 Kubernetes网络模型
 - 11.1.3 Pod网络的实现方式
 - 11.1.4 CNI插件及其常见的实现
 - 11.2 flannel网络插件
 - 11.2.1 flannel的配置参数
 - 11.2.2 VxLAN后端和direct routing
 - 11.2.3 host-gw后端
 - 11.3 网络策略
 - 11.3.1 网络策略概述
 - 11.3.2 部署Canal提供网络策略功能

- 11.3.3 配置网络策略
 - 11.3.4 管控进站流量
 - 11.3.5 管控出站流量
 - 11.3.6 隔离名称空间
 - 11.3.7 网络策略应用案例
 - 11.4 Calico网络插件
 - 11.4.1 Calico工作特性
 - 11.4.2 Calico系统架构
 - 11.4.3 Calico部署要点
 - 11.4.4 部署Calico提供网络服务和网络策略
 - 11.4.5 客户端工具calicoctl
 - 11.5 本章小结
- 第12章 Pod资源调度
- 12.1 Kubernetes调度器概述
 - 12.1.1 常用的预选策略
 - 12.1.2 常用的优选函数
 - 12.2 节点亲和调度
 - 12.2.1 节点硬亲和性
 - 12.2.2 节点软亲和性
 - 12.3 Pod资源亲和调度
 - 12.3.1 位置拓扑
 - 12.3.2 Pod硬亲和调度
 - 12.3.3 Pod软亲和调度
 - 12.3.4 Pod反亲和调度
 - 12.4 污点和容忍度
 - 12.4.1 定义污点和容忍度
 - 12.4.2 管理节点的污点
 - 12.4.3 Pod对象的容忍度
 - 12.4.4 问题节点标识
 - 12.5 Pod优先级和抢占式调度
 - 12.6 本章小结
- 第13章 Kubernetes系统扩展
- 13.1 自定义资源类型 (CRD)
 - 13.1.1 创建CRD对象
 - 13.1.2 自定义资源格式验证
 - 13.1.3 子资源
 - 13.1.4 使用资源类别

- 13.1.5 多版本支持
 - 13.1.6 自定义控制器基础
 - 13.2 自定义API Server
 - 13.2.1 自定义API Server概述
 - 13.2.2 APIService对象
 - 13.3 Kubernetes集群高可用
 - 13.3.1 etcd高可用
 - 13.3.2 Controller Manager和Scheduler高可用
 - 13.4 Kubernetes的部署模式
 - 13.4.1 关键组件
 - 13.4.2 常见的部署模式
 - 13.5 容器时代的DevOps概述
 - 13.5.1 容器：DevOps协作的基础
 - 13.5.2 泛型端到端容器应用程序生命周期 workflow
 - 13.5.3 基于Kubernetes的DevOps
 - 13.6 本章小结
- 第14章 资源指标及HPA控制器
- 14.1 资源监控及资源指标
 - 14.1.1 资源监控及Heapster
 - 14.1.2 新一代监控架构
 - 14.2 资源指标及其应用
 - 14.2.1 部署metrics-server
 - 14.2.2 kubectl top命令
 - 14.3 自定义指标与Prometheus
 - 14.3.1 Prometheus概述
 - 14.3.2 部署Prometheus监控系统
 - 14.3.3 自定义指标适配器k8s-prometheus-adapter
 - 14.4 自动弹性缩放
 - 14.4.1 HPA概述
 - 14.4.2 HPA (v1) 控制器
 - 14.4.3 HPA (v2) 控制器
 - 14.5 本章小结
- 第15章 Helm程序包管理器
- 15.1 Helm基础
 - 15.1.1 Helm的核心术语
 - 15.1.2 Helm架构
 - 15.1.3 安装Helm Client

- 15.1.4 安装Tiller server
- 15.1.5 Helm快速入门
- 15.2 Helm Charts
 - 15.2.1 Charts文件组织结构
 - 15.2.2 Chart.yaml文件组织格式
 - 15.2.3 Charts中的依赖关系
 - 15.2.4 模板和值
 - 15.2.5 其他需要说明的话题
 - 15.2.6 自定义Charts
- 15.3 Helm实践：部署EFK日志管理系统
 - 15.3.1 ElasticSearch集群
 - 15.3.2 日志采集代理fluentd
 - 15.3.3 可视化组件Kibana
- 15.4 本章小结
- 附录A 部署Kubernetes集群
- 附录B 部署GlusterFS及Heketi

前言

为什么要写这本书

作为置身于IT技术领域多年的实践者和教育者，我们一直盼望着行业迎来这样一个时刻：异构的IT基础设施环境所造成的开发和部署系统应用纷繁复杂的局面终于迎来了终结者，开发人员无须再考虑复杂多样的运行环境下软件程序的移植问题，运维人员不用再手动解决运行环境中组件间的依赖关系等，从而让各自的核心职责都回归到开发和保证系统稳定运行本身。终于，以Docker为首的容器技术为此带来了基础保障，并在容器编排技术的支撑下尘埃落定，甚至连IT管理者心心念念多年的DevOps文化运动也借此找到了易于落地的实现方案。于是，系统运行割据多年的局面终于将走向天下一统。

尽管距Kubernetes 1.0的发布不过三四年的光景，但其如今的影响力在IT技术领域完全算得上空前绝后，目前，一众大小公司都在使用或正筹划使用这一IT技术发展史上可能最为成功的开源项目。Linux软件基金会的常务董事Jim Zemlin在Google Cloud Next 17大会上曾表示，Kubernetes是“云时代的Linux”。的确，Kubernetes应该是开源世界有史以来迭代最快的项目，而且在几乎所有需要采用容器技术的场景里成为占统治地位的解决方案，其发展速度恐怕也仅有Linux内核项目可堪匹敌。2018年3月，Kubernetes成为CNCF旗下“毕业”的第一个项目，并荣获2018年OSCON最具影响力奖项。

目前，Kubernetes保持着每年发布四个重要版本的节奏，版本的每次更新都会引入数个新特性。这种快速迭代的机制在为用户不断带来惊喜的同时，也给他们在学习和使用上造成了一些困扰：相关领域的可参考书籍仍不丰富，互联网上可以得到的众多文档并非源于同一个版本，以及厘清脉络拼凑成完整的知识框架所需的时间成本较大。因此，我在课程以及直接或间接参与生产或测试环境的交付之余便萌生了撰写一本Kubernetes入门、进阶与实战的书籍的想法，将自己学习和使用的经验总结、沉淀并分享给更多有此需求的技术同行，帮助大家快速找到入门路径，降低时间成本，并迅速投入测试和生产之用。

的确，在写作过程中，Kubernetes这种快速迭代的机制，以及每每引入的新特性，在小惊喜之余带给笔者更多的却是真真切切的梦魇般的恐惧感：在一年多的写作时间里，许多章节几易其稿，却也依然无法确保能够涵盖即将成为核心功能的特性，于是沮丧感几度如影随形，直到自我安慰着“基础的核心特性基本不会发生大的变动，只要能帮助读者弄清楚Kubernetes系统的基础架构及核心工作逻辑就算工夫没有白费”之后方才释然。于是便有了这本力图尽量多地包罗Kubernetes系统目前主流特性及实践路径的入门和进阶之书、工具之书。

本书特色

本书致力于帮助容器编排技术的初级和中级用户循序渐进地理解与使用Kubernetes系统，因此本书的编写充分考虑到初学者进入新知识领域时的茫然，采用由浅入深、提纲挈领、再由点到面的方式讲解每一个知识细节。对于每个知识点，不仅介绍了其概念和用法，还分析了为什么要有这个概念，实现的方式是什么，背后的逻辑为何，等等，使读者不仅能知其然，还能知其所以然。

本书不仅要带领读者入门，更是一本可以随时动手加以验证的实践手册，而且对于部分重要的内容还会专门一步步地给出具体的实操案例，帮助读者在实践中升华对概念的理解。本书几乎涵盖了应用Kubernetes系统的所有主流知识点，它甚至可以作为计划考取CKA认证的读者的配套参考图书。

读者对象

- 云计算工程师
- 运维工程师
- 系统开发工程师
- 程序架构师
- 计划考取CKA认证的人员

·其他对容器编排感兴趣的人员

如何阅读本书

阅读使用本书之前，读者需要具备Docker容器技术的基础使用能力。本书逻辑上共分为五大部分，15章。

第一部分（第1～2章），介绍Kubernetes系统的基础概念及其基本应用。

第1章 介绍容器编排系统出现的背景，以及Kubernetes系统的功能、特性、核心概念、系统组件及应用模型。

第2章 讲解Kubernetes的核心对象，以及直接使用命令管理资源对象的快速入门技巧。

第二部分（第3～6章），介绍核心资源类型及其应用。

第3章 介绍资源管理模型、陈述式与声明式资源管理接口，并通过命令对比说明两种操作方式的不同之处。

第4章 介绍Pod资源的常用配置、生命周期、存储状态和就绪状态检测，以及计算资源的需求及限制等。

第5章 介绍Pod控制器资源类型，重点讲解了控制无状态应用的ReplicaSet、Deployment、DaemonSet控制器，并介绍了Job和CronJob控制器。

第6章 介绍Service和Ingress资源类型，涵盖Service类型、功用及其实现，以及Ingress控制器、Ingress资源的种类及其实现，并通过案例详细说明了Ingress资源的具体使用方式。

第三部分（第7～9章），介绍存储卷及StatefulSet控制器。

第7章 主要介绍存储卷类型及常见存储卷的使用方式、PV和PVC出现的原因及应用，以及存储类资源的应用和存储卷的动态供给。

第8章 介绍使用一等资源类型ConfigMap和Secret为容器应用提供配置及敏感信息的方式。

第9章 主要介绍有状态应用的Pod控制器资源StatefulSet，包括基础应用、动态扩缩容及更新机制等。

第四部分（第10～11章），介绍安全相关的话题，主要涉及认证、授权、准入控制、网络模型与网络策略。

第10章 重点讲解认证方式、Service Account和TLS认证、授权插件类型及RBAC，并于章节的最后介绍LimitRanger、ResourceQuota和PodSecurityPolicy三种类型的准入控制器及相关的资源类型。

第11章 主要介绍网络插件基础及flannel的三种后端实现与应用、借助Canal插件实现网络策略的方式，以及Calico网络插件的基础使用。

第五部分（第12～15章），介绍Kubernetes系统的高级话题。

第12章 介绍Pod资源的调度策略及高级调度方式的应用，包括节点亲和、Pod资源亲和以及基于污点和容忍度的调度。

第13章 介绍系统资源的扩展方式，包括自定义资源类型、自定义资源对象、自定义API及控制器、Master节点的高可用、基于Kubernetes的PaaS系统等话题。

第14章 介绍资源指标、自定义指标、监控系统及HPA控制器的应用。

第15章 介绍简化应用管理的工具Helm，并基于Helm介绍如何为Kubernetes系统提供统一的日志收集与管理工具栈EFK。

有一定Kubernetes使用经验的读者可以挑选感兴趣的章节阅读。而对于初学者，建议从基础部分逐章阅读，但构建在Kubernetes系统之上的应用多数都要求读者能熟练使用相关领域的知识和技能，如果对某些内容的理解比较困难，那么可能是由于相关知识欠缺，建议读者通过其他资料补充学习相关知识后再阅读本书。编撰本书的主要意

图是为初学者提供一个循序渐进的实操手册，不过，任何读者也都可以将它作为一本案头的工具书随时进行查阅。

排版约定

本书中所有的命令都附带了或长格式或短格式的命令提示符，以便读者区分文中正常使用的“#”和“\$”，命令提示格式为“~]#”或“~]\$”，较长的命令使用了“\”为续行符，且命令及其输出使用了有别于正文的字体。

更多内容和附带代码

本书相关的配置清单等都放置于<https://github.com/ikubernetes/>的相关仓库中，在实践中需要时可直接克隆至本地实验环境中使用。

勘误和支持

尽管进行授课及技术写作已十数年，但动笔著书尚属首次，考虑到排版印刷后的表述无可更改，整个写作过程战战兢兢、如履薄冰。进行每一个关键话题的表述之前都查阅了大量资料，并且反复斟酌，既期望能够将知识点清晰、准确地加以描述，也试图避免因自己的理解偏差而误导读者。尽管如此，由于笔者水平有限，加之编写时间仓促，书中难免存在不妥之处，恳请读者批评指正。如果读者有更多的宝贵意见，请通过邮箱mage@mageedu.com联系我，期待能够得到你们的真挚反馈，在技术之路上互勉共进。另外，本书的勘误将会发布在笔者的博客（<http://www.ilinux.io>）或本书专用的GitHub主页（<https://github.com/ikubernetes>）上，欢迎读者朋友们关注并留言讨论。

参考资料

对于具有不同知识基础和结构的读者来说，仅凭一本书的内容根本不足以获取所需的全部信息，大家还可以通过以下信息获取关于

Kubernetes系统的更多资料，本书在写作期间也从这些参考资料中获得了很大的帮助。

- Kubernetes Documentation和Kubernetes API Reference，这是提供Kubernetes领域相关知识最全面、最深入和最准确的参考材料。

- 《Kubernetes in Action》，本书的谋篇布局及写作理念与此书不谋而合，因此笔者在本书中对许多概念的理解和验证也以此书为素材，并且在写作之时有多处概念的描述也借鉴了此书的内容。

- Red Hat Openshift Documentation，Red Hat公司的产品文档规范、权威、细致且条理清晰，是不可多得的参考材料。

- The New Stack的技术文章及调研报告是了解Kubernetes系统技术细节和行业应用现状及趋势的不可多得的优秀资源。

- Bitnami及Heptio站点上的博客文章提供了深入了解和学习Kubernetes系统某个特定技术细节的可靠资料。

另外，本书还大量借鉴了通过搜索引擎所获取到的不少技术文章和参考文档，在这里一并向这些书籍和文章的作者表示深深的谢意！

致谢

感谢Kubernetes社区创造性的劳动成果和辛苦付出，我们因此有了学习和使用如此优秀的开源系统的可能性，这也是本书得以编撰的基石。

感谢我的同事们在我写作期间给予的支持和理解，他们的努力让我拥有了得以放心写作的时间和精力。感谢提供了相关行业信息并促使我下定决心开始编写此书的张常安和张士杰先生，本书的写作过程中也得到了他们支援的宝贵素材。

感谢参加了我的课程（马哥教育）的学员朋友们，大家的学习热情及工作中源源不断反馈而来的信息与需求在不同程度上帮助我一直保持着对技术的追求和热忱，教学相长在此得到了充分的体现。

感谢机械工业出版社华章公司高婧雅女士对本书写作的悉心指导，以及对我本人的包容和理解。

最后要特别感谢我的家人，为写作这本书，我牺牲了很多陪伴他们的时间，也正因为他们在生活中的关怀和鼓励才使我能够踏踏实实地完成本书内容的编写。

马永亮

2018年10月

第1章 Kubernetes系统基础

近十几年来，IT领域新技术、新概念层出不穷，例如DevOps、微服务（Microservice）、容器（Container）、云计算（Cloud Computing）和区块链（Blockchain）等，直有“乱花渐欲迷人眼”之势。另外，出于业务的需要，IT应用模型也在不断地变革，例如，开发模式从瀑布式（Waterfall）到敏捷（Agile）再到精益（Lean），甚至是与QA和Operations融合的DevOps，应用程序架构从单体（monolithic）模型到分层模型再到微服务，部署及打包方式从面向物理机到虚拟机再到容器，应用程序的基础架构从自建机房到托管再到云计算，等等，这些变革使得IT技术应用的效率大大提升，同时却以更低的成本交付更高质量的产品。

尤其是以Docker为代表的容器技术的出现，终结了DevOps中交付和部署环节因环境、配置及程序本身的不同而造成的动辄几种甚至十几种部署配置的困境，将它们统一在容器镜像（image）之上。如今，越来越多的企业或组织开始选择以镜像文件作为交付载体。容器镜像之内直接包含了应用程序及其依赖的系统环境、库、基础程序等，从而能够在容器引擎上直接运行。于是，IT运维工程师（operator）无须关注开发应用程序的编程语言、环境配置等，甚至连业务逻辑本身也不必过多关注，而只需要掌握容器管理的单一工具链即可。

部署的复杂度虽然降低了，但以容器格式运行的应用程序间的协同却成了一个新的亟待解决的问题，这种需求在微服务架构中表现得尤为明显。结果，以Kubernetes为代表的容器编排系统应需而生。

1.1 容器技术概述

容器是一种轻量级、可移植、自包含的软件打包技术，它使得应用程序可以在几乎任何地方以相同的方式运行。软件开发工程师在自己笔记本上创建并测试完成的容器，无须任何修改就能够在生产系统的虚拟机、物理机或云主机上运行。

容器由应用程序本身和它的环境依赖（库和其他应用程序）两部分组成，并在宿主机（**Host**）操作系统的用户空间中运行，但与操作系统的其他进程互相隔离，它们的实现机制有别于诸如**VMWare**、**KVM**和**Xen**等实现方案的传统虚拟化技术。容器与虚拟机的对比关系如图1-1所示。

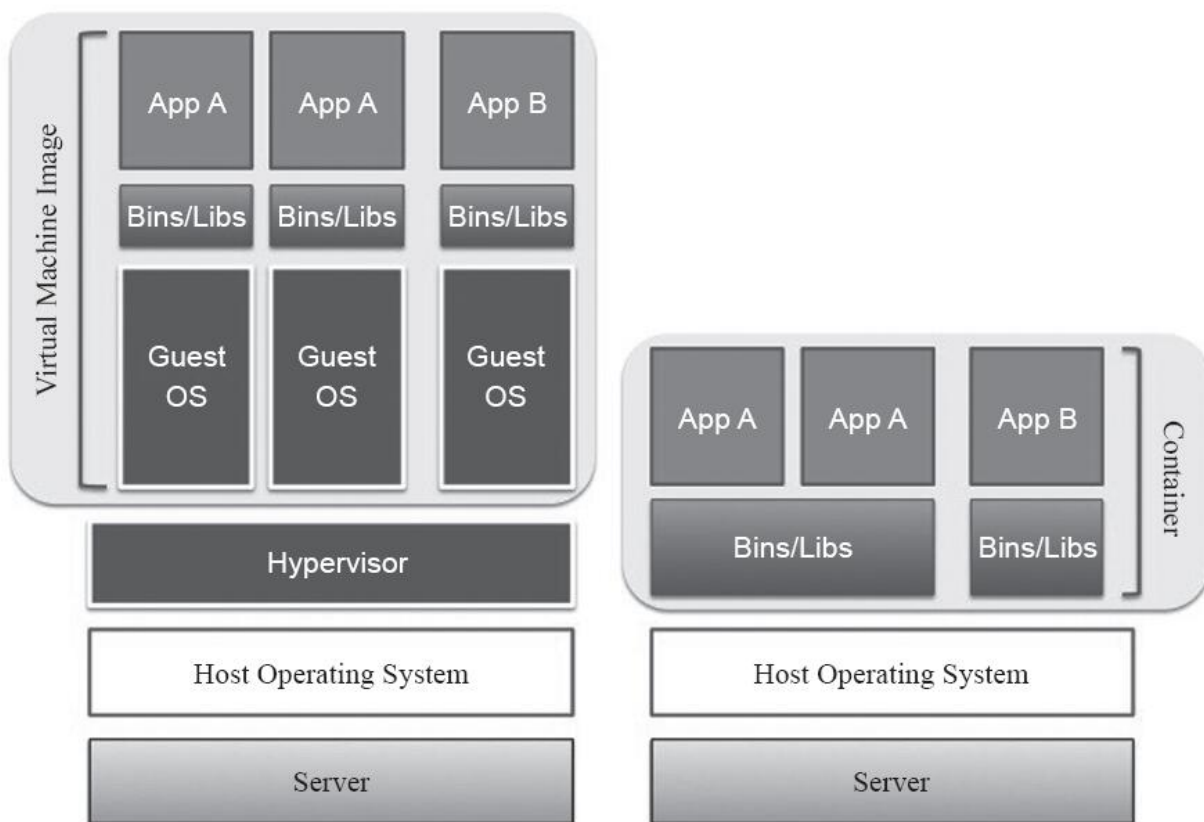


图1-1 容器和虚拟机对比

(<http://www.nuagenetworks.net/blog/containers/>)

由于同一个宿主机上的所有容器都共享其底层操作系统（内核空间），这就使得容器在体积上要比传统的虚拟机小得多。另外，启动容器无须启动整个操作系统，所以容器部署和启动的速度更快，开销更小，也更容易迁移。事实上，容器赋予了应用程序超强的可移植能力。

1.1.1 容器技术的功用

IT系统在架构上已经迭代数十年之久，其环境复杂程度日趋加重，直有积重难返之势。现如今，应用程序开发人员通常需要同时使用多种服务构建，并要架构IT信息系统，涉及MQ、Cache和DB等，且很可能要部署到不同的环境中，如物理服务器、虚拟服务器、私有云或公有云之上。这些不同的主机或许还有着不同的系统环境，如RHEL、Debian或SUSE等Linux发行版，甚至是UNIX、Windows等。

结果，一方面应用程序包含了多种服务，每种服务均可能存在依赖的库和软件包；另一方面存在多种部署环境，而服务在运行时又可能需要动态迁移到不同的环境中。于是，各种服务和环境通过排列组合产生了一个大部署矩阵。应用程序开发工程师在编写代码时需要考虑不同的运行环境，而运维工程师则需要为不同的服务和平台配置环境。对他们双方来说，这都必将是一项困难而艰巨的任务。

幸运的是，货运系统的集装箱机制为解决这个难题提供了有效的借鉴方案。**Docker**正是将集装箱思想运用到软件打包上，为代码提供了一个基于容器的标准化运输系统。**Docker**可以将几乎任何应用程序及其依赖的运行环境都打包成一个轻量级、可移植、自包含的容器，并能够运行于支持**Docker**容器引擎的所有操作系统之上。简言之，容器的优势主要表现在以下两个方面。

·应用程序开发工程师：“一次构建，到处运行”（**Build Once, Run Anywhere**）。容器意味着环境隔离和可重复性，开发人员只需为应用创建一个运行环境，并将其打包成容器便可在各种部署环境上运行，并与它所在的宿主机环境隔离。

·运维工程师：“一次配置，运行所有”（**Configure Once, Run Anything**）。一旦配置好标准的容器运行时环境，服务器就可以运行任何容器，这使得运维人员的工作变得更高效、一致和可重复。容器消除了开发、测试、生产环境的不一致性。

1.1.2 容器简史

容器技术的概念最初出现在2000年，当时称为FreeBSD jail，这种技术可将FreeBSD系统分区为多个子系统（也称为Jail）。2001年，通过Jacques Gélinas的VServer项目，隔离环境的实施理念进入了Linux领域。

Jail的目的是让进程在经过修改的chroot环境中创建，而不会脱离和影响整个系统——chroot环境对文件系统、网络和用户的访问都实现了虚拟化。然而，Jail在实施方面存在着不少的局限性，当它与Namespaces和CGroups等技术结合在一起之后，才让这种隔离方法从构想变为了现实。后来，Linux容器项目（LXC）又为其添加了一些用户常用的工具、模板、库和语言绑定，从而较好地改善了用户使用容器技术时的体验。

Docker在LXC项目的基础上，从文件系统、网络互联到进程隔离等方面对容器技术进行了进一步的封装，极大地简化了容器的创建和维护过程，从而促进了容器技术的大流行。Docker最初是由dotCloud公司创始人Solomon Hykes在法国期间发起的一个公司内部项目，并于2013年3月以Apache 2.0授权协议开源，其项目代码托管于GitHub之上。虽然其最初的实现是基于LXC项目的，但Docker在后来的0.7版本转为使用自行开发的libcontainer容器引擎，而1.11版本又将其换作了runC和containerd。



提示 在2017年4月举行的DockerCon上，Docker公司将GitHub上原本隶属于Docker组织的Docker项目直接转移到了一个新的名为Moby的组织下，并将其重命名为Moby项目。

1.1.3 Docker的功能限制

Docker本身非常适合用于管理单个容器，不过，一旦开始使用越来越多的容器封装和运行应用程序，必将会导致其管理和编排变得越来越困难。最终，用户不得不对容器实施分组，以便跨所有容器提供网络、安全、监控等服务。于是，以**Kubernetes**为代表的容器编排系统应运而生。

真正的生产型应用会涉及多个容器，这些容器必须跨多个服务器主机进行部署。**Kubernetes**可以提供所需的编排和管理功能，以使用户针对这些工作负载轻松完成大规模容器部署。而且，借助于**Kubernetes**的编排功能，用户可以构建出跨多个容器的应用服务，并且可以实现跨集群调度、扩展容器，以及长期持续管理这些容器的健康状况等。使用中，**Kubernetes**还需要与网络、存储、安全性、监控及其他服务进行整合，以提供全面的容器基础架构，如图1-2所示。

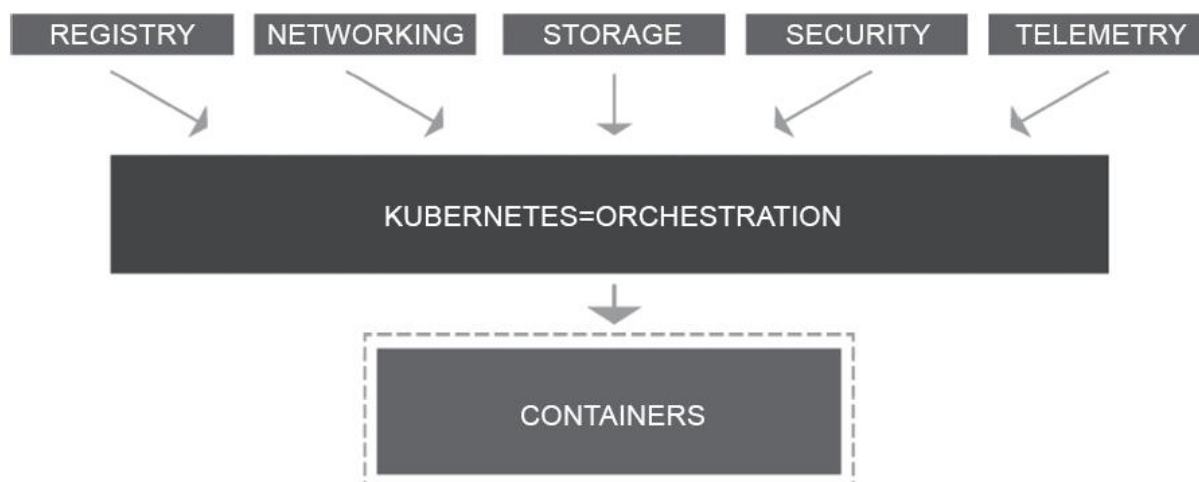


图1-2 容器与容器编排（来源：RedHat Inc.）

Kubernetes利用容器的扩缩容机制解决了许多常见的问题，它将容器归类到一起，形成“容器集”（Pod），为分组的容器增加了一个抽象层，用于帮助用户调度工作负载（workload），并为这些容器提供所需的联网和存储等服务。**Kubernetes**的其他部分可帮助用户在这些Pod之间达成负载均衡，同时确保运行正确数量的容器，以充分支持实际的工作负载。

1.2 Kubernetes概述

尽管公开面世不过短短数年时间，**Kubernetes**业已成为容器编排领域事实上的标准，其近一两年的发展状态也在不断地验证着**Urs Hlzle**曾经的断言：无论是公有云、私有云抑或混合云，**Kubernetes**都将作为一个为任何应用、任何环境提供的容器管理框架而无处不在。

1.2.1 Kubernetes简史

Kubernetes（来自希腊语，意为“舵手”或“飞行员”）由Joe Beda、Brendan Burns和Craig McLuckie创立，而后Google的其他几位工程师，包括Brian Grant和Tim Hockin等加盟共同研发，并由Google在2014年首次对外宣布。Kubernetes的开发和设计都深受Google内部系统Borg的影响，事实上，它的许多顶级贡献者之前也是Borg系统的开发者。

Borg是Google内部使用的大规模集群管理系统，久负盛名。它建构于容器技术之上，目的是实现资源管理的自动化，以及跨多个数据中心的资源利用率最大化。2015年4月，Borg论文《Large-scale cluster management at Google with Borg》伴随Kubernetes的高调宣传被Google首次公开，人们终于有缘得窥其全貌。

事实上，正是由于诞生于容器世家Google，并站在Borg这个巨人的肩膀之上，充分受益于Borg过去十数年间积累的经验和教训，Kubernetes甫一面世就立即广受关注和青睐，并迅速称霸了容器编排技术领域。很多人将Kubernetes视为Borg系统的一个开源实现版本，在Google内部，Kubernetes的原始代号曾经是Serven of Nine，即星际迷航中友好的“Borg”角色，它标识中的舵轮有七个轮辐就是对该项目代号的致意，如图1-3所示。



图1-3 Kubernetes Logo

Kubernetes v1.0于2015年7月21日发布，紧随其后，Google与Linux基金会合作组建了Cloud Native Computing Foundation（云原生计算基金会，简称为CNCF），并将Kubernetes作为种子技术予以提供。这之后，Kubernetes进入了版本快速迭代期，从此不断地融入着新功能，如Federation、Network Policy API、RBAC、CRD和CSI，等等，并增加了对Windows系统的支持。

2017年可谓是容器生态发展史上具有里程碑意义的一年。这一年，AWS、Azure和Alibaba Cloud都相继在其原有容器服务上新增了对Kubernetes的支持，而Docker官方也在2017年10月宣布同时支持Swarm和Kubernetes编排系统。这一年，RKT容器派系的CoreOS舍弃掉自己的调度工具Fleet，将其商用平台Tectonic的重心转移至Kubernetes。这一年，Mesos也于9月宣布了对Kubernetes的支持，其平台用户可以安装、扩展和升级多个生产级的Kubernetes集群。这一年，Rancher Labs推出了其2.0版本的容器管理平台并宣布all-in Kubernetes，放弃了其内置多年的容器编排系统Cattle。类似的故事依然在进行，并且必将在一个时期内持续上演。

1.2.2 Kubernetes特性

Kubernetes是一种用于在一组主机上运行和协同容器化应用程序的系统，旨在提供可预测性、可扩展性与高可用性的方法来完全管理容器化应用程序和服务的生命周期的平台。用户可以定义应用程序的运行方式，以及与其他应用程序或外部世界交互的途径，并能实现服务的扩容和缩容，执行平滑滚动更新，以及在不同版本的应用程序之间调度流量以测试功能或回滚有问题的部署。**Kubernetes**提供了接口和可组合的平台原语，使得用户能够以高度的灵活性和可靠性定义及管理应用程序。简单总结起来，它具有以下几个重要特性。

(1) 自动装箱

建构于容器之上，基于资源依赖及其他约束自动完成容器部署且不影响其可用性，并通过调度机制混合关键型应用和非关键型应用的工作负载于同一节点以提升资源利用率。

(2) 自我修复（自愈）

支持容器故障后自动重启、节点故障后重新调度容器，以及其他可用节点、健康状态检查失败后关闭容器并重新创建等自我修复机制。

(3) 水平扩展

支持通过简单命令或UI手动水平扩展，以及基于CPU等资源负载率的自动水平扩展机制。

(4) 服务发现和负载均衡

Kubernetes通过其附加组件之一的KubeDNS（或CoreDNS）为系统内置了服务发现功能，它会为每个Service配置DNS名称，并允许集群内的客户端直接使用此名称发出访问请求，而Service则通过iptables或ipvs内建了负载均衡机制。

(5) 自动发布和回滚

Kubernetes支持“灰度”更新应用程序或其配置信息，它会监控更新过程中应用程序的健康状态，以确保它不会在同一时刻杀掉所有实例，而此过程中一旦有故障发生，就会立即自动执行回滚操作。

（6）密钥和配置管理

Kubernetes的ConfigMap实现了配置数据与Docker镜像解耦，需要时，仅对配置做出变更而无须重新构建Docker镜像，这为应用开发部署带来了很大的灵活性。此外，对于应用所依赖的一些敏感数据，如用户名和密码、令牌、密钥等信息，Kubernetes专门提供了Secret对象为其解耦，既便利了应用的快速开发和交付，又提供了一定程度上的安全保障。

（7）存储编排

Kubernetes支持Pod对象按需自动挂载不同类型的存储系统，这包括节点本地存储、公有云服务商的云存储（如AWS和GCP等），以及网络存储系统（例如，NFS、iSCSI、GlusterFS、Ceph、Cinder和Flocker等）。

（8）批量处理执行

除了服务型应用，Kubernetes还支持批处理作业及CI（持续集成），如果需要，一样可以实现容器故障后恢复。

1.2.3 Kubernetes概念和术语

Kubernetes使用共享网络将多个物理机或虚拟机汇集到一个集群中，在各服务器之间进行通信，该集群是配置Kubernetes的所有组件、功能和工作负载的物理平台。集群中一台服务器（或高可用部署中的一组服务器）用作**Master**，负责管理整个集群，余下的其他机器用作**Worker Node**（早期版本中也称为**Minion**），它们是使用本地和外部资源接收和运行工作负载的服务器，如图1-4所示。集群中的这些主机可以是物理服务器，也可以是虚拟机（包括IaaS云端的VPS）。

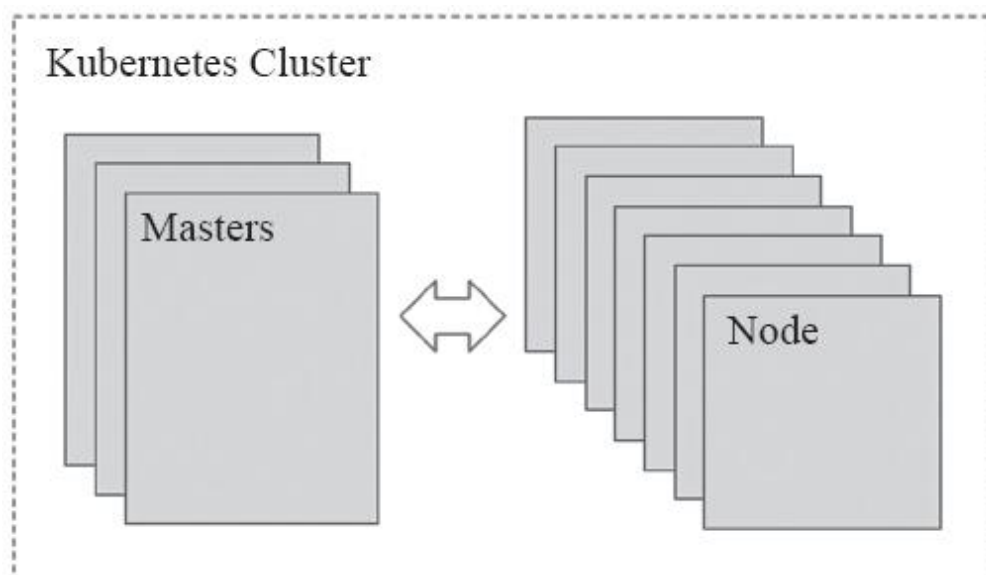


图1-4 Kubernetes集群主机

(1) Master

Master是集群的网关和中枢，负责诸如为用户和客户端暴露API、跟踪其他服务器的健康状态、以最优方式调度工作负载，以及编排其他组件之间的通信等任务，它是用户或客户端与集群之间的核心联络点，并负责Kubernetes系统的大多数集中式管控逻辑。单个**Master**节点即可完成其所有的功能，但出于冗余及负载均衡等目的，生产环境中通常需要协同部署多个此类主机。**Master**节点类似于蜂群中的蜂王。

(2) Node

Node是Kubernetes集群的工作节点，负责接收来自**Master**的工作指令并根据指令相应地创建或销毁**Pod**对象，以及调整网络规则以合理地路由和转发流量等。理论上讲，**Node**可以是任何形式的计算设备，不过**Master**会统一将其抽象为**Node**对象进行管理。**Node**类似于蜂群中的工蜂，生产环境中，它们通常数量众多。

Kubernetes将所有**Node**的资源集结于一处形成一台更加强大的“服务器”，如图1-5所示，在用户将应用部署于其上时，**Master**会使用调度算法将其自动指派至某个特定的**Node**运行。在**Node**加入集群或从集群中移除时，**Master**也会按需重新编排影响到的**Pod**（容器）。于是，用户无须关心其应用究竟运行于何处。

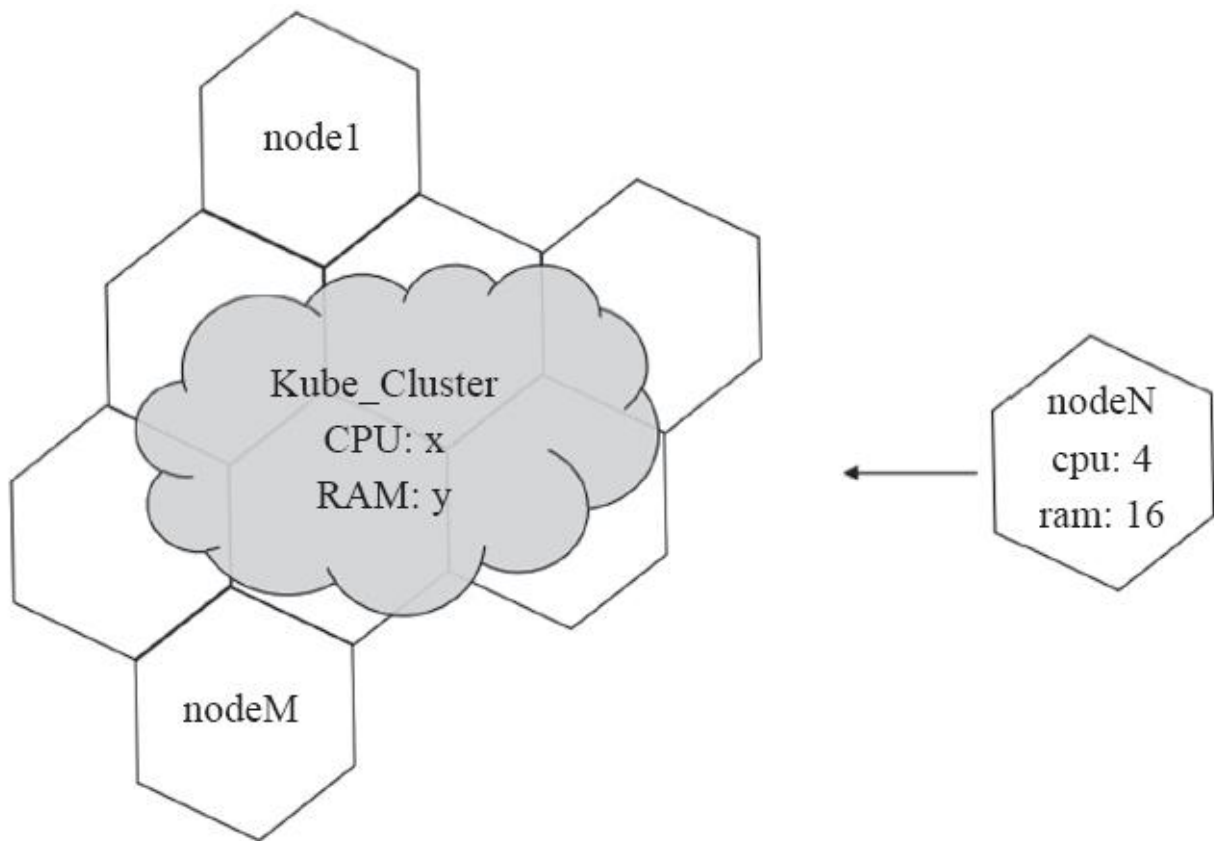


图1-5 Node组成的虚拟资源池

从抽象的视角来讲，**Kubernetes**还有着众多的组件来支撑其内部的业务逻辑，包括运行应用、应用编排、服务暴露、应用恢复等，它们在**Kubernetes**中被抽象为**Pod**、**Service**、**Controller**等资源类型，下面列出了几个较为常用的资源抽象。

(1) Pod

Kubernetes并不直接运行容器，而是使用一个抽象的资源对象来封装一个或者多个容器，这个抽象即为Pod，它也是Kubernetes的最小调度单元。同一Pod中的容器共享网络名称空间和存储资源，这些容器可经由本地回环节口lo直接通信，但彼此之间又在Mount、User及PID等名称空间上保持了隔离。尽管Pod中可以包含多个容器，但是作为最小调度单元，它应该尽可能地保持“小”，即通常只应该包含一个主容器，以及必要的辅助型容器（sidecar），如图1-6所示。

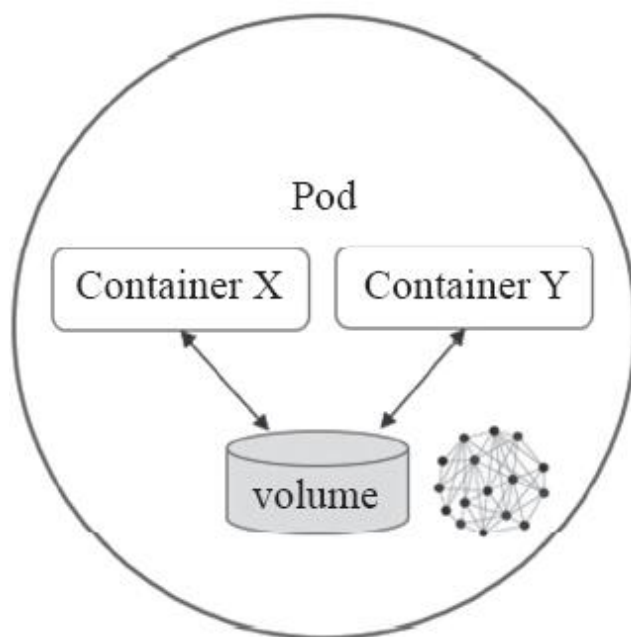


图1-6 Kubernetes Pod示意图

(2) 资源标签

标签（Label）是将资源进行分类的标识符，资源标签其实就是一个键值型（key/values）数据。标签旨在指定对象（如Pod等）辨识性的属性，这些属性仅对用户存在特定的意义，对Kubernetes集群来说并不直接表达核心系统语义。标签可以在对象创建时附加其上，并能够在创建后的任意时间进行添加和修改。一个对象可以拥有多个标签，一个标签也可以附加于多个对象（通常是同一类对象）之上，如图1-7所示。

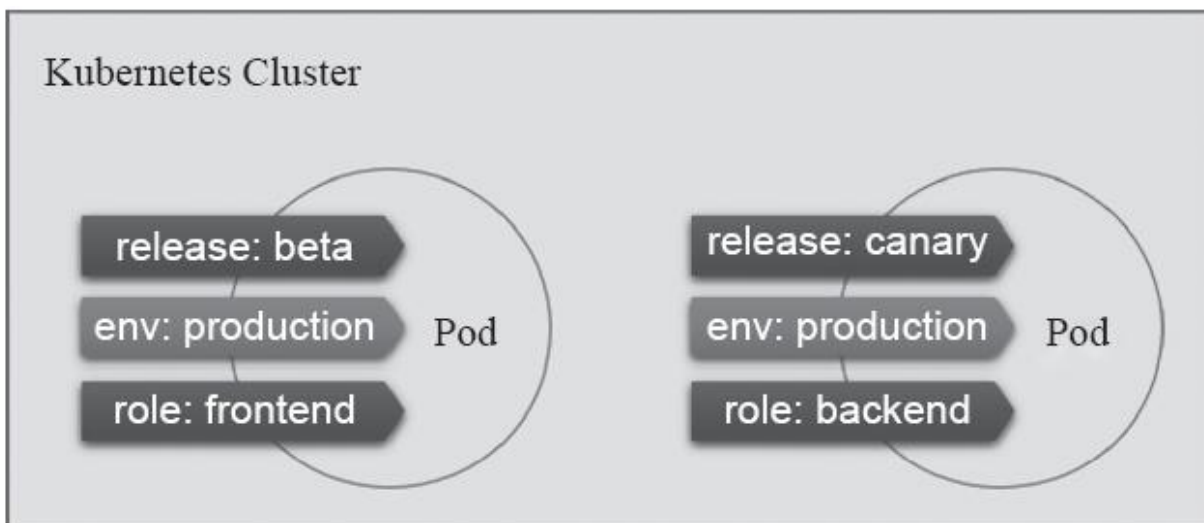


图1-7 Kubernetes资源标签

(3) 标签选择器

标签选择器（Selector）全称为“Label Selector”，它是一种根据Label来过滤符合条件的资源对象的机制。例如，将附有标签“role: backend”的所有Pod对象挑选出来归为一组就是标签选择器的一种应用，如图1-8所示。用户通常使用标签对资源对象进行分类，而后使用标签选择器挑选出它们，例如将其创建为某Service的端点。

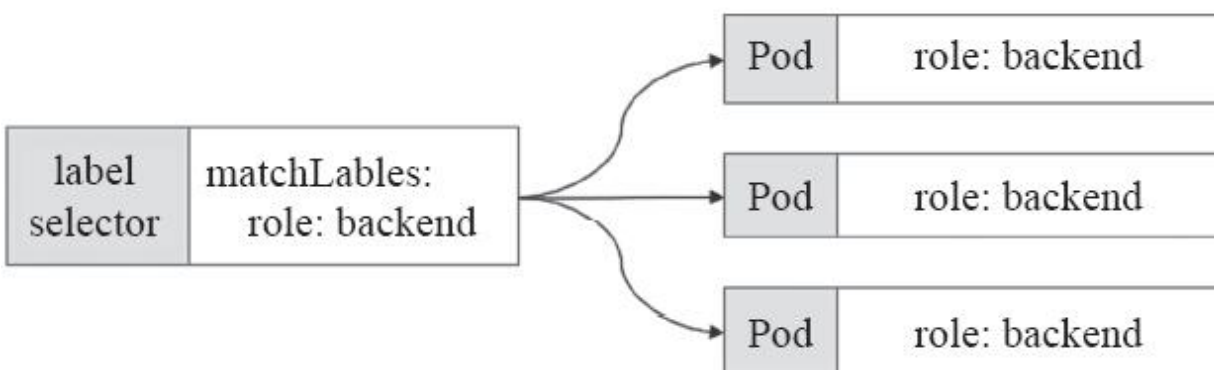


图1-8 标签选择器

(4) Pod控制器

尽管Pod是Kubernetes的最小调度单元，但用户通常并不会直接部署及管理Pod对象，而是要借助于另一类抽象—控制器（Controller）对其进行管理。用于工作负载的控制器是一种管理Pod生命周期的资源抽

象，它们是Kubernetes上的一类对象，而非单个资源对象，包括ReplicationController、ReplicaSet、Deployment、StatefulSet、Job等。以图1-9中所示的Deployment控制器为例，它负责确保指定的Pod对象的副本数量精确符合定义，否则“多退少补”。使用控制器之后就不再需要手动管理Pod对象了，用户只需要声明应用的期望状态，控制器就会自动对其进行进程管理。

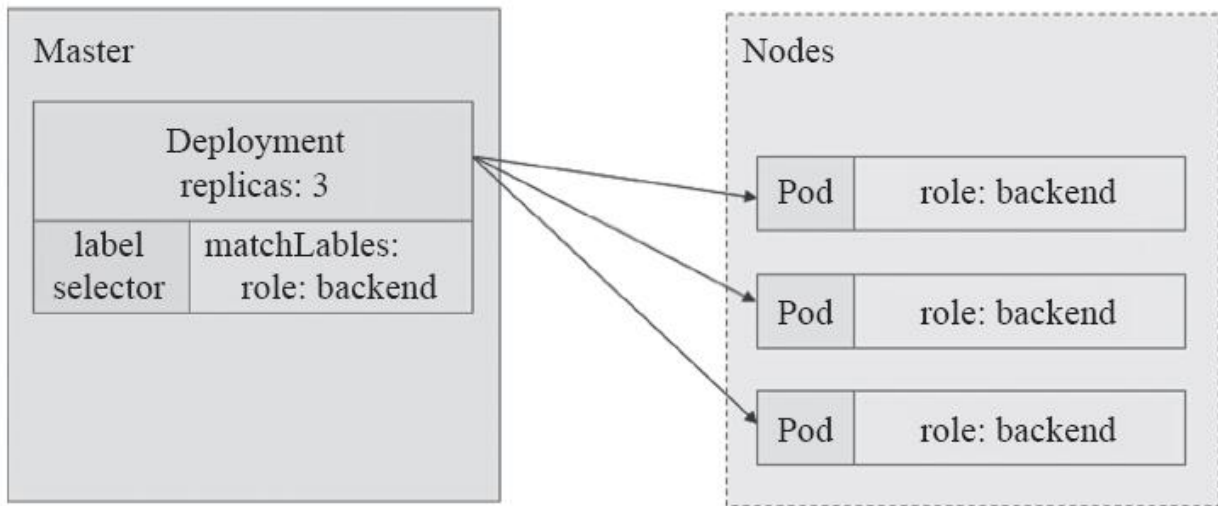


图1-9 Deployment控制器示意图

(5) 服务资源 (Service)

Service是建立在一组Pod对象之上的资源抽象，它通过标签选择器选定一组Pod对象，并为这组Pod对象定义一个统一的固定访问入口（通常是一个IP地址），若Kubernetes集群存在DNS附件，它就会在Service创建时为其自动配置一个DNS名称以便客户端进行服务发现。到达Service IP的请求将被负载均衡至其后的端点——各个Pod对象之上，因此Service从本质上来讲是一个四层代理服务。另外，Service还可以将集群外部流量引入到集群中来。

(6) 存储卷

存储卷 (Volume) 是独立于容器文件系统之外的存储空间，常用于扩展容器的存储空间并为它提供持久存储能力。Kubernetes集群上的存储卷大体可分为临时卷、本地卷和网络卷。临时卷和本地卷都位于Node本地，一旦Pod被调度至其他Node，此种类型的存储卷将无法访

问到，因此临时卷和本地卷通常用于数据缓存，持久化的数据则需要放置于持久卷（persistent volume）之上。

（7）Name和Namespace

名称（Name）是Kubernetes集群中资源对象的标识符，它们的作用域通常是名称空间（Namespace），因此名称空间是名称的额外的限定机制。在同一个名称空间中，同一类型资源对象的名称必须具有唯一性。名称空间通常用于实现租户或项目的资源隔离，从而形成逻辑分组，如图1-10所示。创建的Pod和Service等资源对象都属于名称空间级别，未指定时，它们都属于默认的名称空间“default”。

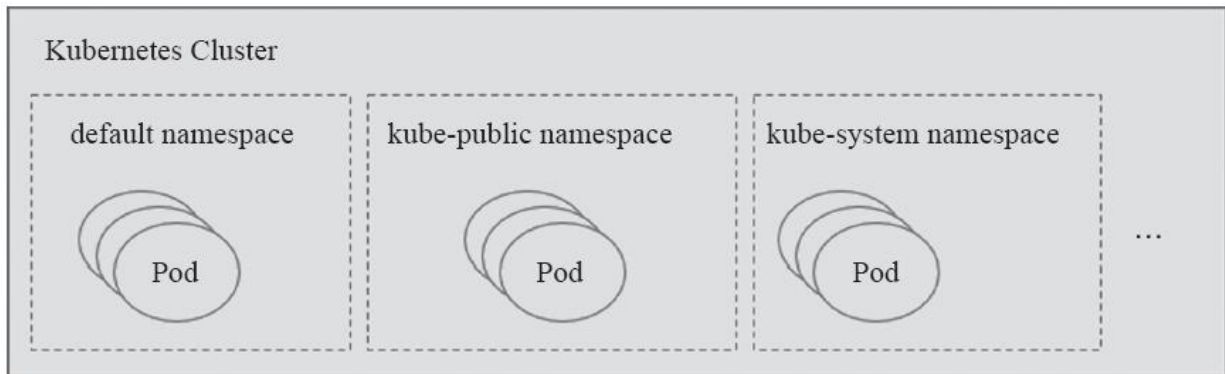


图1-10 名称空间

（8）Annotation

Annotation（注解）是另一种附加在对象之上的键值类型的数据，但它拥有更大的数据容量。Annotation常用于将各种非标识型元数据（metadata）附加到对象上，但它不能用于标识和选择对象，通常也不会被Kubernetes直接使用，其主要目的是方便工具或用户的阅读及查找等。

（9）Ingress

Kubernetes将Pod对象和外部网络环境进行了隔离，Pod和Service等对象间的通信都使用其内部专用地址进行，如若需要开放某些Pod对象提供给外部用户访问，则需要为其请求流量打开一个通往Kubernetes集群内部的通道，除了Service之外，Ingress也是这类通道的实现方式之一。

1.3 Kubernetes集群组件

一个典型的Kubernetes集群由多个工作节点（**worker node**）和一个集群控制平面（**control plane**，即**Master**），以及一个集群状态存储系统（**etcd**）组成。其中**Master**节点负责整个集群的管理工作，为集群提供管理接口，并监控和编排集群中的各个工作节点。各节点负责以**Pod**的形式运行容器，因此，各节点需要事先配置好容器运行依赖到的所有服务和资源，如容器运行时环境等。Kubernetes的系统架构如图1-11所示。

Master节点主要由**apiserver**、**controller-manager**和**scheduler**三个组件，以及一个用于集群状态存储的**etcd**存储服务组成，而每个**Node**节点则主要包含**kubelet**、**kube-proxy**及容器引擎（**Docker**是最为常用的实现）等组件。此外，完整的集群服务还依赖于一些附加组件，如**KubeDNS**等。

1.3.1 Master组件

Kubernetes的集群控制平面由多个组件组成，这些组件可统一运行于单一Master节点，也可以以多副本的方式同时运行于多个节点，以为Master提供高可用功能，甚至还可以运行于Kubernetes集群自身之上。Master主要包含以下几个组件。

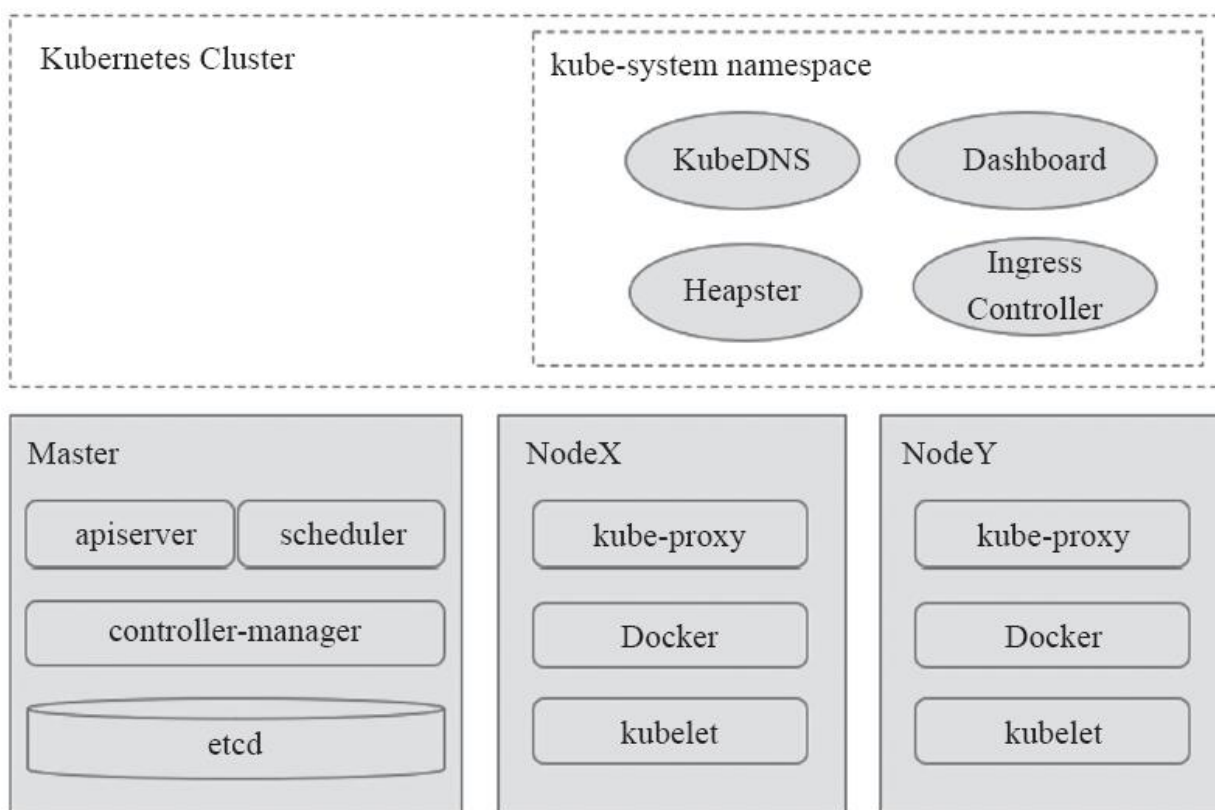


图1-11 Kubernetes系统组件

(1) API Server

API Server负责输出RESTful风格的Kubernetes API，它是发往集群的所有REST操作命令的接入点，并负责接收、校验并响应所有的REST请求，结果状态被持久存储于etcd中。因此，API Server是整个集群的网关。

(2) 集群状态存储（Cluster State Store）

Kubernetes集群的所有状态信息都需要持久存储于存储系统etcd中，不过，etcd是由CoreOS基于Raft协议开发的分布式键值存储，可用于服务发现、共享配置以及一致性保障（如数据库主节点选择、分布式锁等）。因此，etcd是独立的服务组件，并不隶属于Kubernetes集群自身。生产环境中应该以etcd集群的方式运行以确保其服务可用性。

etcd不仅能够提供键值数据存储，而且还为其提供了监听（watch）机制，用于监听和推送变更。Kubernetes集群系统中，etcd中的键值发生变化时会通知到API Server，并由其通过watch API向客户端输出。基于watch机制，Kubernetes集群的各组件实现了高效协同。

（3）控制器管理器（Controller Manager）

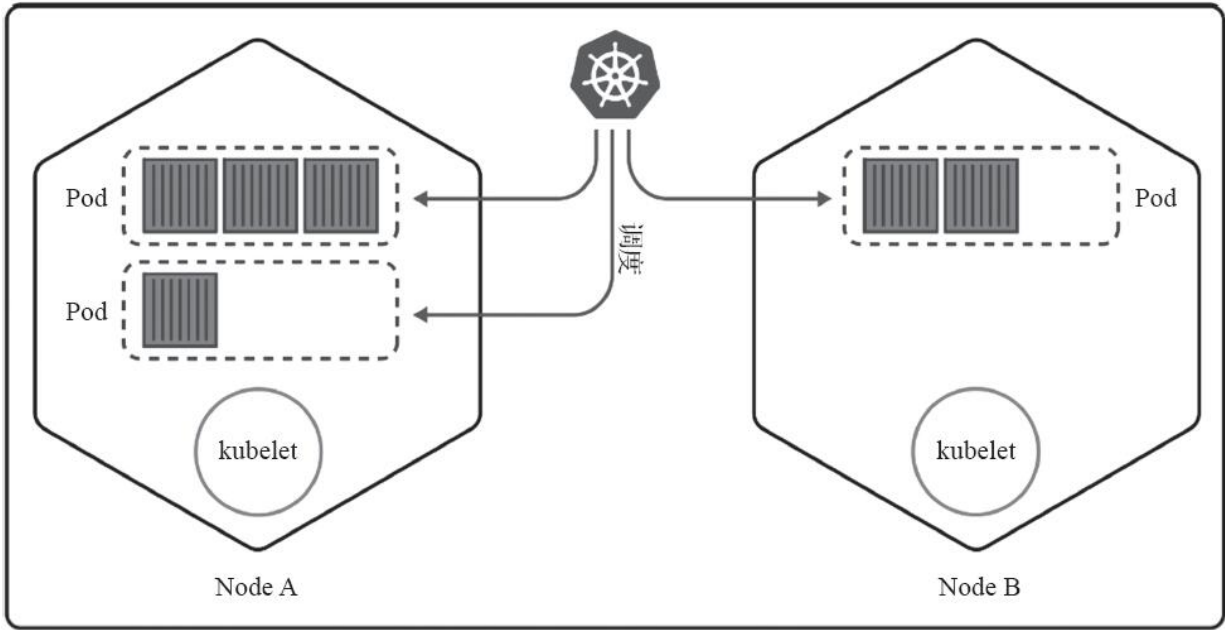
Kubernetes中，集群级别的大多数功能都是由几个被称为控制器的进程执行实现的，这几个进程被集成于kube-controller-manager守护进程中。由控制器完成的功能主要包括生命周期功能和API业务逻辑，具体如下。

- 生命周期功能：包括Namespace创建和生命周期、Event垃圾回收、Pod终止相关的垃圾回收、级联垃圾回收及Node垃圾回收等。

- API业务逻辑：例如，由ReplicaSet执行的Pod扩展等。

（4）调度器（Scheduler）

Kubernetes是用于部署和管理大规模容器应用的平台，根据集群规模的不同，其托管运行的容器很可能会数以千计甚至更多。API Server确认Pod对象的创建请求之后，便需要由Scheduler根据集群内各节点的可用资源状态，以及要运行的容器的资源需求做出调度决策，其工作逻辑如图1-12所示。另外，Kubernetes还支持用户自定义调度器。



集群

图1-12 Kubernetes调度器

1.3.2 Node组件

Node负责提供运行容器的各种依赖环境，并接受Master的管理。每个Node主要由以下几个组件构成。

(1) Node的核心代理程序kubelet

kubelet是运行于工作节点之上的守护进程，它从API Server接收关于Pod对象的配置信息并确保它们处于期望的状态（desired state，后文不加区别地称之为“目标状态”）。kubelet会在API Server上注册当前工作节点，定期向Master汇报节点资源使用情况，并通过cAdvisor监控容器和节点的资源占用状况。

(2) 容器运行时环境

每个Node都要提供一个容器运行时（Container Runtime）环境，它负责下载镜像并运行容器。kubelet并未固定链接至某容器运行时环境，而是以插件的方式载入配置的容器环境。这种方式清晰地定义了各组件的边界。目前，Kubernetes支持的容器运行环境至少包括Docker、RKT、cri-o和Fraki等。

(3) kube-proxy

每个工作节点都需要运行一个kube-proxy守护进程，它能够按需为Service资源对象生成iptables或ipvs规则，从而捕获访问当前Service的ClusterIP的流量并将其转发至正确的后端Pod对象。

1.3.3 核心附件

Kubernetes集群还依赖于一组称为“附件”（add-ons）的组件以提供完整的功能，它们通常是由第三方提供的特定应用程序，且托管运行于Kubernetes集群之上，如图1-11所示。下面列出的几个附件各自为集群从不同角度引用了所需的核心功能。

- KubeDNS**: 在Kubernetes集群中调度运行提供DNS服务的Pod，同一集群中的其他Pod可使用此DNS服务解决主机名。Kubernetes自1.11版本开始默认使用CoreDNS项目为集群提供服务注册和服务发现的动态名称解析服务，之前的版本中用到的是kube-dns项目，而SkyDNS则是更早一代的项目。

- Kubernetes Dashboard**: Kubernetes集群的全部功能都要基于Web的UI，来管理集群中的应用甚至是集群自身。

- Heapster**: 容器和节点的性能监控与分析系统，它收集并解析多种指标数据，如资源利用率、生命周期事件等。新版本的Kubernetes中，其功能会逐渐由Prometheus结合其他组件所取代。

- Ingress Controller**: Service是一种工作于传统层的负载均衡器，而Ingress是在应用层实现的HTTP（s）负载均衡机制。不过，Ingress资源自身并不能进行“流量穿透”，它仅是一组路由规则的集合，这些规则需要通过Ingress控制器（Ingress Controller）发挥作用。目前，此类的可用项目有Nginx、Traefik、Envoy及HAProxy等。

1.4 Kubernetes网络模型基础

云计算的核心是虚拟化技术，网络虚拟化技术又是其最重要的组成部分，用于在物理网络上虚拟多个相互隔离的虚拟网络，实现网络资源切片，提高网络资源利用率，实现弹性化网络。Kubernetes作为容器云技术栈中的容器编排组件，必然需要在多租户（名称空间）的基础上实现弹性网络管理，这也是“基础设施即代码”的要求之一。

1.4.1 网络模型概述

Kubernetes的网络中主要存在四种类型的通信：同一Pod内的容器间通信、各Pod彼此之间的通信、Pod与Service间的通信，以及集群外部的流量同Service之间的通信。Kubernetes为Pod和Service资源对象分别使用了各自的专用网络，Pod网络由Kubernetes的网络插件配置实现，而Service的网络则由Kubernetes集群予以指定。为了提供更灵活的解决方式，Kubernetes的网络模型需要借助于外部插件实现，它要求任何实现机制都必须满足以下需求。

- 所有Pod间均可不经NAT机制而直接通信。
- 所有节点均可不经NAT机制而直接与所有容器通信。
- 容器自己使用的IP也是其他容器或节点直接看到的地址。换句话说，所有Pod对象都位于同一平面网络中，而且可以使用Pod自身的地址直接通信。

Kubernetes使用的网络插件必须能为Pod提供满足以上要求的网络，它需要为每个Pod配置至少一个特定的地址，即Pod IP。Pod IP地址实际存在于某个网卡（可以是虚拟设备）上，而Service的地址却是一个虚拟IP地址，没有任何网络接口配置此地址，它由kube-proxy借助iptables规则或ipvs规则重新定向到本地端口，再将其调度至后端Pod对象。Service的IP地址是集群提供服务的接口，也称为Cluster IP。

Pod网络及其IP由Kubernetes的网络插件负责配置和管理，具体使用的网络地址可在管理配置网络插件时指定，如10.244.0.0/16网络。而Cluster网络和IP则是由Kubernetes集群负责配置和管理，如10.96.0.0/12网络。

总结起来，Kubernetes集群至少应该包含三个网络，如图1-13中的网络环境所示。一个是各主机（Master、Node和etcd等）自身所属的网络，其地址配置于主机的网络接口，用于各主机之间的通信，例如，Master与各Node之间的通信。此地址配置于Kubernetes集群构建之前，它并不能由Kubernetes管理，管理员需要于集群构建之前自行确定其地址配置及管理方式。第二个是Kubernetes集群上专用于Pod资源对象的

网络，它是一个虚拟网络，用于为各Pod对象设定IP地址等网络参数，其地址配置于Pod中容器的网络接口之上。Pod网络需要借助kubenetwork插件或CNI插件实现，该插件可独立部署于Kubernetes集群之外，亦可托管于Kubernetes之上，它需要在构建Kubernetes集群时由管理员进行定义，而后在创建Pod对象时由其自动完成各网络参数的动态配置。第三个是专用于Service资源对象的网络，它也是一个虚拟网络，用于为Kubernetes集群之中的Service配置IP地址，但此地址并不配置于任何主机或容器的网络接口之上，而是通过Node之上的kube-proxy配置为iptables或ipvs规则，从而将发往此地址的所有流量调度至其后端的各Pod对象之上。Service网络在Kubernetes集群创建时予以指定，而各Service的地址则在用户创建Service时予以动态配置。

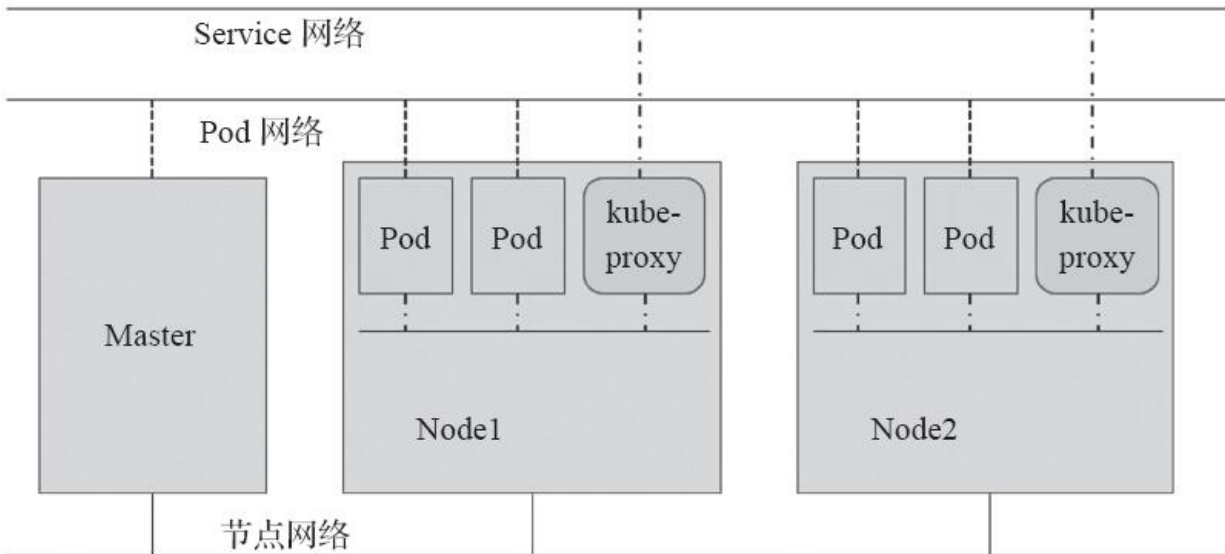


图1-13 Kubernetes网络环境



提示 CNI是指容器网络接口（Container Network Interface），是由CNCF（Cloud Native Computing Foundation）维护的项目，其由一系列的用于编写配置容器网络插件的规范和库接口（libcni）组成，支持众多插件项目。后文对此有详细说明。

1.4.2 集群上的网络通信

Kubernetes集群的客户端大体可以分为两类：API Server客户端和应用程序（运行为Pod中的容器）客户端，如图1-14所示。第一类客户端通常包含人类用户和Pod对象两种，它们通过API Server访问Kubernetes集群完成管理任务，例如，管理集群上的各种资源对象。第二类客户端一般也包含人类用户和Pod对象两种，它们的访问目标是Pod上运行于容器中的应用程序提供的各种具体的服务，如redis或nginx等，不过，这些访问请求通常要经由Service或Ingress资源对象进行。另外，第二类客户端的访问目标对象的操作要经由第一类客户端创建和配置完成后才能进行。

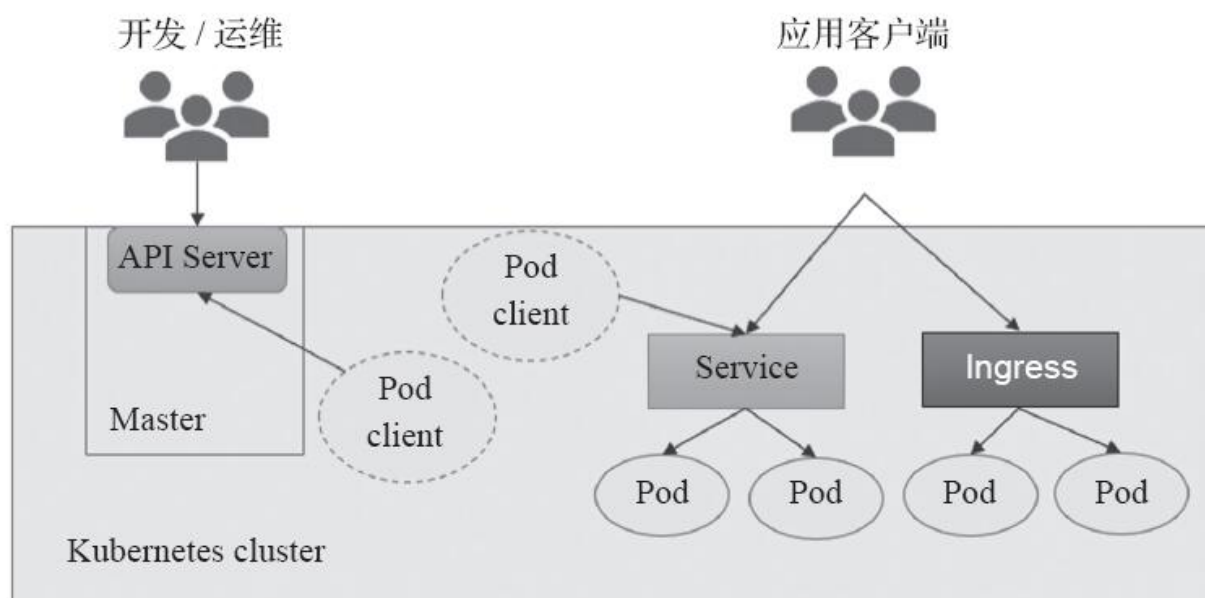


图1-14 Kubernetes客户端及其类型

访问API Server时，人类用户一般借助于命令行工具kubectl或图形UI（例如Kubernetes Dashboard）进行，也可通过编程接口进行访问，包括REST API。访问Pod中的应用时，其访问方式要取决于Pod中的应用程序，例如，对于运行Nginx容器的Pod来说，其最常用工具自然是浏览器。

管理员（开发人员或运维人员）使用Kubernetes集群的常见操作包括通过控制器创建Pod，在Pod的基础上创建Service供第二类客户端访

问，更新Pod中的应用版本（更新和回滚）以及对应用规模进行扩容或缩容等，另外还有集群附件管理、存储卷管理、网络及网络策略管理、资源管理和安全管理等，这些内容将在后面的章节中展开。不过，这一切的前提是要先构建出一个可用的Kubernetes集群，这一内容将在第2章中着重讲述。

1.5 本章小结

本章介绍了Kubernetes的历史、功用、特性及其相关的核心概述和术语，并简单描述了其架构及其各关键组件，以及集群网络中的常见通信方式，具体如下。

- Kubernetes集群主要由Master和Node两类节点组成。

- Master主要包含API Server、controller-manager、Scheduler和etcd几个组件，其中API Server是整个集群的网关。

- Node主要由kubelet、kube-proxy和容器引擎等组件构成，kubelet是Kubernetes集群的工作于节点之上的代理组件。

- 完整的Kubernetes集群还需要部署有CoreDNS（或KubeDNS）、Prometheus（或Heapster）、Dashboard和Ingress Controller几个附加组件。

- Kubernetes的网络中主要存在四种类型的通信：同一Pod内的容器间通信、各Pod间的通信、Pod与Service间的通信，以及集群外部的流量同Service之间的通信。

第2章 Kubernetes快速入门

Kubernetes集群将所有节点上的资源都整合到一个大的虚拟资源池里，以代替一个个单独的服务器，而后开放诸如CPU、内存和I/O这些基本资源用于运行其基本单元—Pod资源对象。Pod的容器中运行着隔离的任务单元，它们以Pod为原子单位，并根据其资源需求从虚拟资源池中为其动态分配资源。若可以将整个集群类比为一台传统的服务器，那么Kubernetes（Master）就好比是操作系统内核，其主要职责在于抽象资源并调度任务，而Pod资源对象就是那些运行于用户空间中的进程。于是，传统意义上的向单节点或集群直接部署、配置应用的模型日渐式微，取而代之的是向Kubernetes的API Server提交运行Pod对象。

API Server是负责接收并响应客户端提交任务的接口，用户可使用诸如CLI工具（如kubectl）、UI工具（如Dashboard）或程序代码（客户端开发库）发起请求，其中，kubectl是最为常用的交互式命令行工具。快速了解Kubernetes的办法之一就是部署一个测试集群，并尝试测试使用它的各项基本功能。本章在简单介绍核心资源对象后将尝试使用kubectl创建Deployment和Service资源部署并暴露一个Web应用，以便读者快速了解如何在Kubernetes系统上运行应用程序的核心任务。

2.1 Kubernetes的核心对象

API Server提供了RESTful风格的编程接口，其管理的资源是Kubernetes API中的端点，用于存储某种API对象的集合，例如，内置Pod资源是包含了所有Pod对象的集合。资源对象是用于表现集群状态的实体，常用于描述应于哪个节点进行容器化应用、需要为其配置什么资源以及应用程序的管理策略等，例如，重启、升级及容错机制。另外，一个对象也是一种“意向记录”——一旦创建，Kubernetes就需要一直确保对象始终存在。Pod、Deployment和Service等都是最常用的核心对象。

2.1.1 Pod资源对象

Pod资源对象是一种集合了一到多个应用容器、存储资源、专用IP及支撑容器运行的其他选项的逻辑组件，如图2-1所示。换言之，Pod代表着Kubernetes的部署单元及原子运行单元，即一个应用程序的单一运行实例，它通常由共享资源且关系紧密的一个或多个应用容器组成。

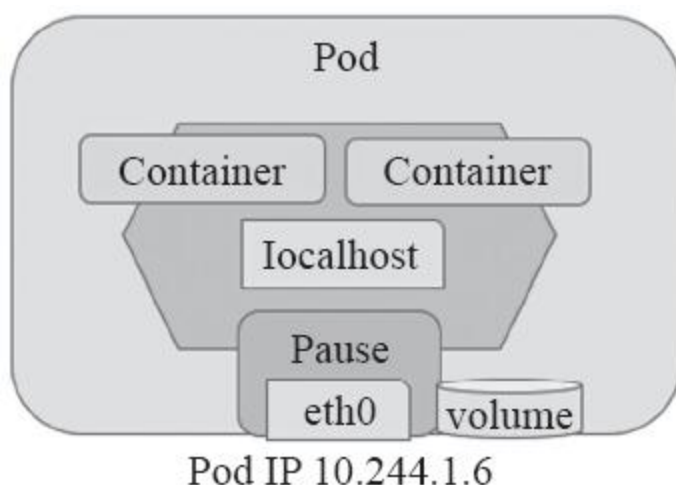


图2-1 Pod通常由一到多个共享网络和存储资源的容器组合而成

Kubernetes的网络模型要求其各Pod对象的IP地址位于同一网络平面内（同一IP网段），各Pod之间可使用其IP地址直接进行通信，无论它们运行于集群内的哪个工作节点之上，这些Pod对象都像是运行于同一局域网中的多个主机。

读者可以将每个Pod对象想象成一个逻辑主机，它类似于现实世界中的物理主机或VM（Virtual Machine），运行于同一个Pod对象中的多个进程也类似于物理机或VM上独立运行的进程。不过，Pod对象中的各进程均运行于彼此隔离的容器中，并于各容器间共享两种关键资源：[网络](#)和[存储卷](#)。

·网络（networking）：每个Pod对象都会被分配一个集群内专用的IP地址，也称为Pod IP，同一Pod内部的所有容器共享Pod对象的Network和UTS名称空间，其中包括主机名、IP地址和端口等。因此，

这些容器间的通信可以基于本地回环接口lo进行，而与Pod外的其他组件的通信则需要使用Service资源对象的ClusterIP及其相应的端口完成。

·存储卷（volume）：用户可以为Pod对象配置一组“存储卷”资源，这些资源可以共享给其内部的所有容器使用，从而完成容器间数据的共享。存储卷还可以确保在容器终止后被重启，甚至是被删除后也能确保数据不会丢失，从而保证了生命周期内的Pod对象数据的持久化存储。

一个Pod对象代表某个应用程序的一个特定实例，如果需要扩展应用程序，则意味着为此应用程序同时创建多个Pod实例，每个实例均代表应用程序的一个运行的“副本”（replica）。这些副本化的Pod对象的创建和管理通常由另一组称之为“控制器”（Controller）的对象实现，例如，Deployment控制器对象。

创建Pod时，还可以使用Pod Preset对象为Pod注入特定的信息，如ConfigMap、Secret、存储卷、卷挂载和环境变量等。有了Pod Preset对象，Pod模板的创建者就无须为每个模板显式提供所有信息，因此，也就无须事先了解需要配置的每个应用的细节即可完成模板定义。这些内容将在后面的章节中予以介绍。

基于期望的目标状态和各节点的资源可用性，Master会将Pod对象调度至某选定的工作节点运行，工作节点于指向的镜像仓库（image registry）下载镜像，并于本地的容器运行时环境中启动容器。Master会将整个集群的状态保存于etcd中，并通过API Server共享给集群的各组件及客户端。

2.1.2 Controller

Kubernetes集群的设计中，Pod是有生命周期的对象。用户通过手工创建或由Controller（控制器）直接创建的Pod对象会被“调度器”（Scheduler）调度至集群中的某工作节点运行，待到容器应用进程运行结束之后正常终止，随后就会被删除。另外，节点资源耗尽或故障也会导致Pod对象被回收。

但Pod对象本身并不具有“自愈”功能，若是因为工作节点甚至是调度器自身导致了运行失败，那么它将会被删除；同样，资源耗尽或节点故障导致的回收操作也会删除相关的Pod对象。在设计上，Kubernetes使用“控制器”实现对一次性的（用后即弃）Pod对象的管理操作，例如，要确保部署的应用程序的Pod副本数量严格反映用户期望的数目，以及基于Pod模板来重建Pod对象等，从而实现Pod对象的扩缩容、滚动更新和自愈能力等。例如，某节点发生故障时，相关的控制器会将此节点上运行的Pod对象重新调度到其他节点进行重建。

控制器本身也是一种资源类型，它有着多种实现，其中与工作负载相关的实现如Replication Controller、Deployment、StatefulSet、DaemonSet和Jobs等，也可统称它们为Pod控制器。如图2-2中的Deployment就是这类控制器的代表实现，是目前最常用的管理无状态应用的Pod控制器。

Pod控制器的定义通常由期望的副本数量、Pod模板和标签选择器（Label Selector）组成。Pod控制器会根据标签选择器对Pod对象的标签进行匹配检查，所有满足选择条件的Pod对象都将受控于当前控制器并计入其副本总数，并确保此数目能够精确反映期望的副本数。

需要注意的是，在实际的应用场景中，在接收到的请求流量负载显著低于或接近于已有Pod副本的整体承载能力时，用户需要手动修改Pod控制器中的期望副本数量以实现应用规模的扩容或缩容。不过，若集群中部署了Heapster或Prometheus一类的资源指标监控附件时，用户还可以使用“HorizontalPodAutoscaler”（HPA）计算出合适的Pod副本数量，并自动修改Pod控制器中期望的副本数以实现应用规模的动态伸缩，提高集群资源利用率，如图2-3所示。

Kubernetes集群中的每个节点都运行着cAdvisor以收集容器及节点的CPU、内存及磁盘资源的利用率指标数据，这些统计数据由Heapster聚合后可通过API Server访问。HorizontalPodAutoscaler基于这些统计数据监控容器健康状态并做出扩展决策。

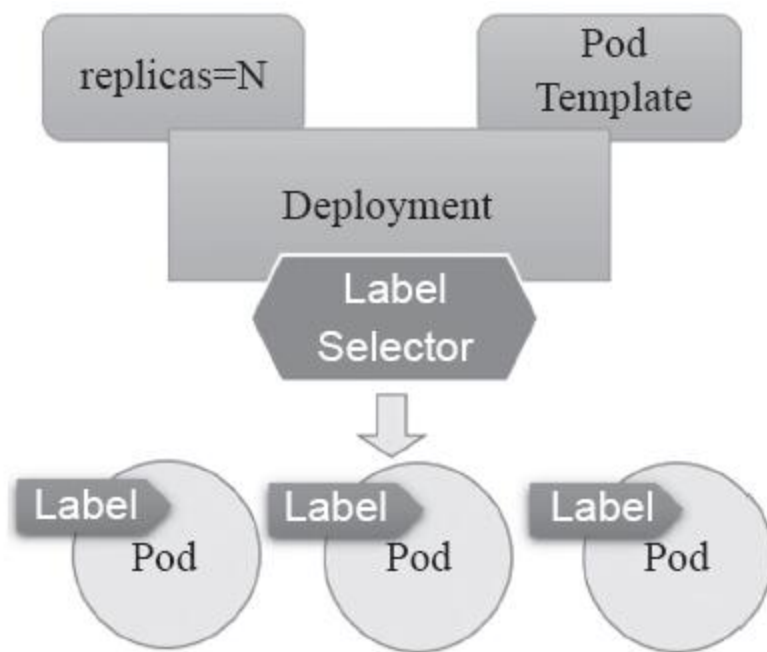


图2-2 Replication Controller

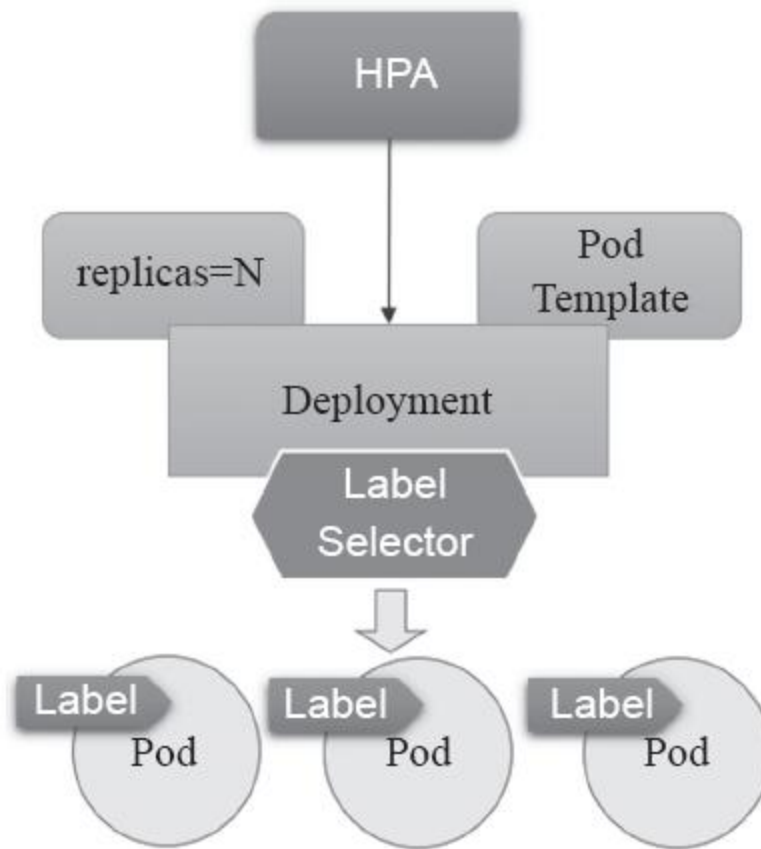


图2-3 Horizontal Pod Autoscaler

2.1.3 Service

尽管Pod对象可以拥有IP地址，但此地址无法确保在Pod对象重启或被重建后保持不变，这会为集群中的Pod应用间依赖关系的维护带来麻烦：前端Pod应用（依赖方）无法基于固定地址持续跟踪后端Pod应用（被依赖方）。于是，Service资源被用于在被访问的Pod对象中添加一个有着固定IP地址的中间层，客户端向此地址发起访问请求后由相关的Service资源调度并代理至后端的Pod对象。

换言之，Service是“微服务”的一种实现，事实上它是一种抽象：通过规则定义出由多个Pod对象组合而成的逻辑集合，并附带访问这组Pod对象的策略。Service对象挑选、关联Pod对象的方式同Pod控制器一样，都是要基于Label Selector进行定义，其示意图如图2-4所示。

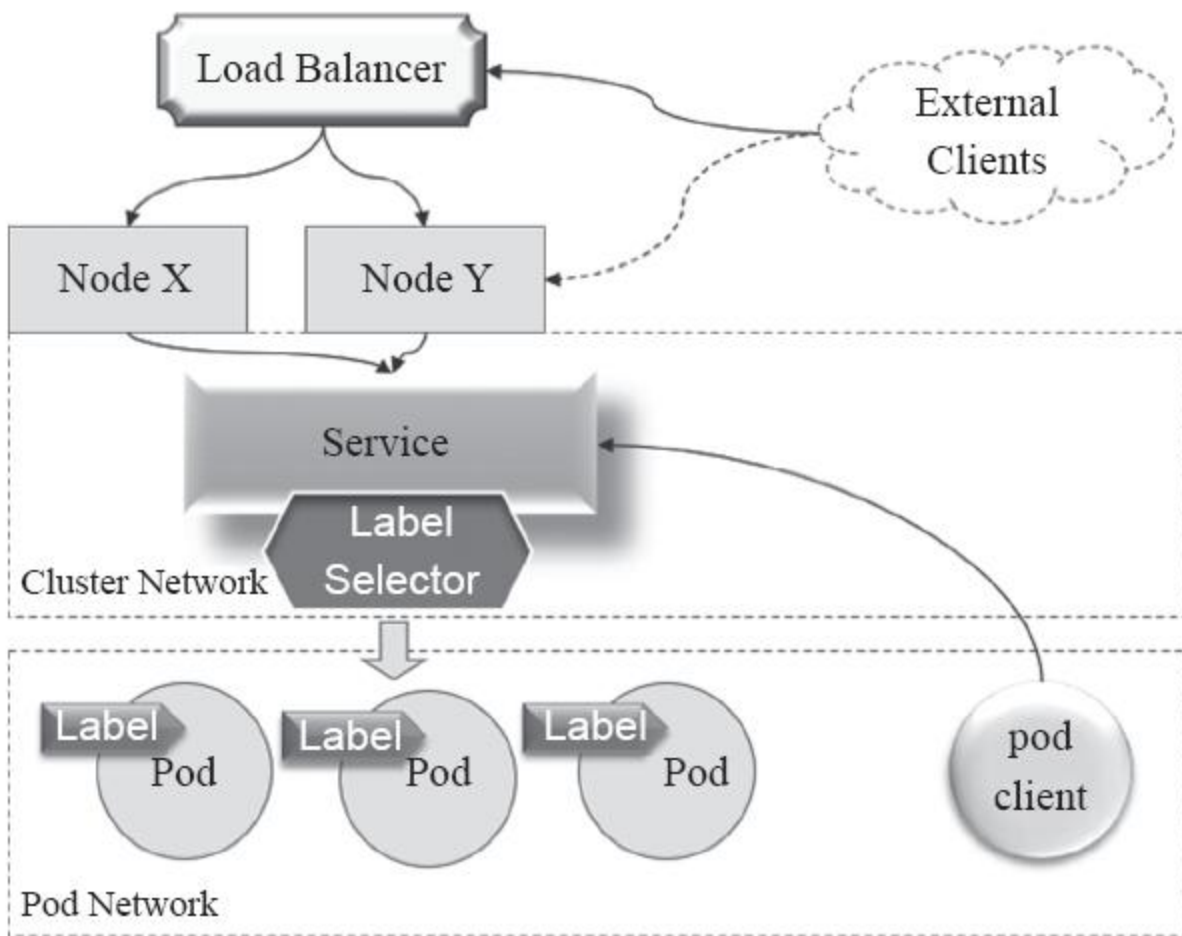


图2-4 Service对象功能示意图

Service IP是一种虚拟IP，也称为**Cluster IP**，它专用于集群内通信，通常使用专用的地址段，如“10.96.0.0/12”网络，各**Service**对象的IP地址在此范围内由系统动态分配。

集群内的**Pod**对象可直接请求此类的**Cluster IP**，例如，图2-4中来自**pod client**的访问请求即可以**Service**的**Cluster IP**作为目标地址，但集群网络属于私有网络地址，它们仅在集群内部可达。将集群外部的访问流量引入集群内部的常用方法是通过节点网络进行，实现方法是通过工作节点的IP地址和某端口（**NodePort**）接入请求并将其代理至相应的**Service**对象的**Cluster IP**上的服务端点，而后由**Service**对象将请求代理至后端的**Pod**对象的**Pod IP**及应用程序监听的端口。因此，诸如图2-4中的**External Clients**这种来自集群外部的客户端无法直接请求此**Service**提供的服务，而是需要事先经由某一个工作节点（如**Node Y**）的IP地址进行，这类请求需要两次转发才能到达目标**Pod**对象，因此在通信效率上必然存在负面影响。

事实上，**NodePort**会部署于集群中的每一个节点，这就意味着，集群外部的客户端通过任何一個工作节点的IP地址来访问定义好的**NodePort**都可以到达相应的**Service**对象。此种场景中，如果存在集群外部的一个负载均衡器，即可将用户请求负载均衡至集群中的部分或者所有节点。这是一种称为“**LoadBalancer**”类型的**Service**，它通常是由**Cloud Provider**自动创建并提供的软件负载均衡器，不过，也可以是由管理员手工配置的诸如**F5Big-IP**一类的硬件设备。

简单来说，**Service**主要有三种常用类型：第一种是仅用于集群内部通信的**ClusterIP**类型；第二种是接入集群外部请求的**NodePort**类型，它工作于每个节点的主机IP之上；第三种是**LoadBalancer**类型，它可以把外部请求负载均衡至多个**Node**的主机IP的**NodePort**之上。此三种类型中，每一种都以其前一种为基础才能实现，而且第三种类型中的**LoadBalancer**需要协同集群外部的组件才能实现，并且此外部组件并不接受Kubernetes的管理。

2.1.4 部署应用程序的主体过程

Docker容器技术使得部署应用程序从传统的安装、配置、启动应用程序的方式转为了容器引擎上基于镜像创建和运行容器，而Kubernetes又使得创建和运行容器的操作不必再关注其位置，并在一定程度上赋予了它动态扩缩容及自愈的能力，从而让用户从主机、系统及应用程序的维护工作中解脱出来。

用到某应用程序时，用户只需要向API Server请求创建一个Pod控制器，由控制器根据镜像等信息向API Server请求创建出一定数量的Pod对象，并由Master之上的调度器指派至选定的工作节点以运行容器化应用。此外，用户一般还需要创建一个具体的Service对象以便为这些Pod对象建立起一个固定的访问入口，从而使得其客户端能够通过其服务名称或ClusterIP进行访问，如图2-5所示。

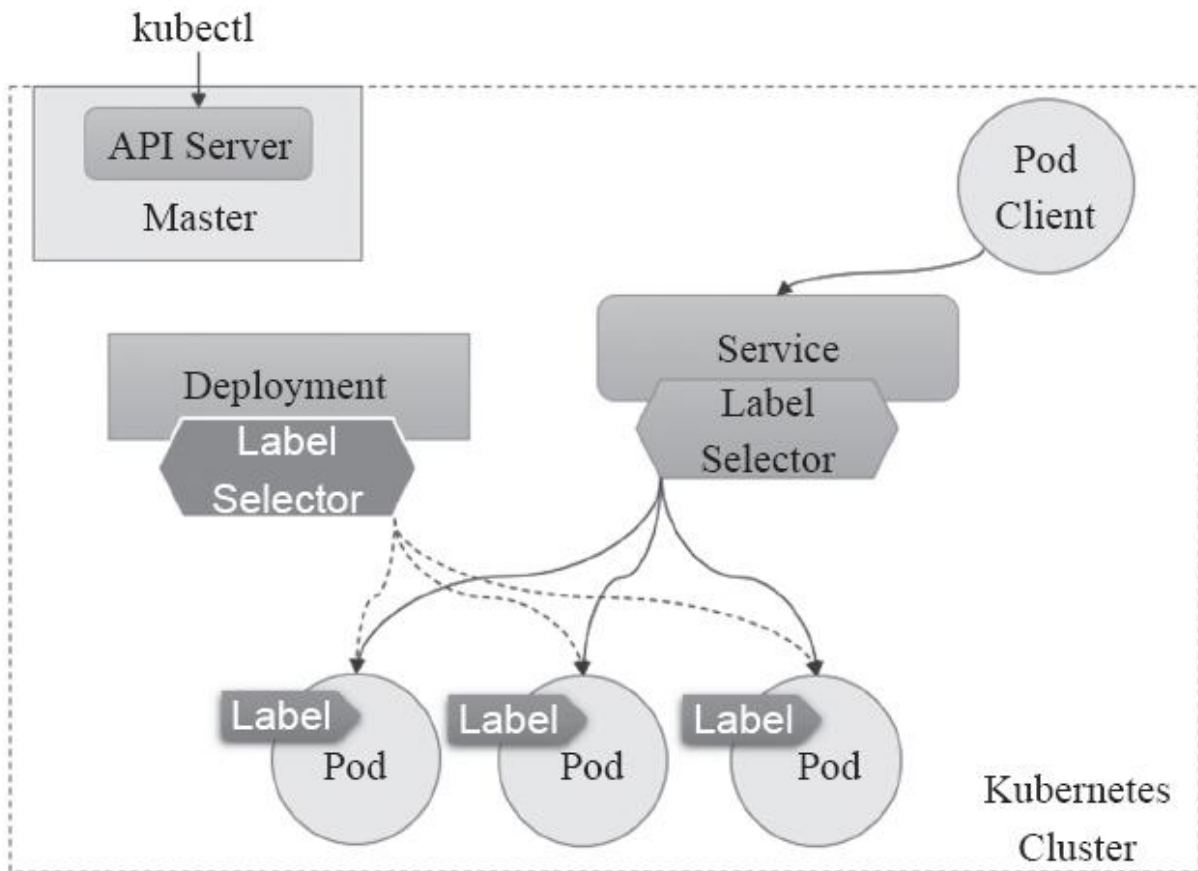


图2-5 应用程序简单的部署示例

API Server的常用客户端程序是Kubernetes系统自带的命令行工具kubectI，它通过一众子命令用于实现集群及相关资源对象的管理操作，并支持直接命令式、命令式配置清单及声明式配置清单等三种操作方式，特性丰富且功能强大。而需作为集群附件额外部署的Dashboard则提供了基于Web界面的图形客户端，它是一个通用目的的管理工具，与Kubernetes紧密集成，支持多级别用户授权，能在一定程度上替代kubectI的大多数操作。

本章后面的篇幅将介绍在部署完成的Kubernetes集群环境中如何快速部署如图2-5所示的示例应用程序，并简单说明如何完成对容器化应用的访问，以及如何应用规模的动态伸缩，并借此让读者了解kubectI命令的基本功能和用法。

2.2 部署Kubernetes集群

Kubernetes系统可运行于多种平台之上，包括虚拟机、裸服务器或PC等，例如本地主机或托管的云端虚拟机。若仅用于快速了解或开发的目的，那么读者可直接于单个主机之上部署“伪”分布式的Kubernetes集群，将集群的所有组件均部署运行于单台主机上，著名的minikube项目可帮助用户快速构建此类环境。如果要学习使用Kubernetes集群的完整功能，则应该构建真正的分布式集群环境，将Master和Node等部署于多台主机之上，主机的具体数量要按实际需求而定。另外，集群部署的方式也有多种选择，简单的可以基于kubeadm一类的部署工具运行几条命令即可实现，而复杂的则可以是从零开始手动构建集群环境。

2.2.1 kubeadm部署工具

kubeadm是Kubernetes项目自带的集群构建工具，它负责执行构建一个最小化的可用集群以及将其启动等的必要基本步骤，简单来讲，kubeadm是Kubernetes集群全生命周期的管理工具，可用于实现集群的部署、升级/降级及拆除，如图2-6所示。不过，在部署操作中，kubeadm仅关心如何初始化并启动集群，余下的其他操作，例如安装Kubernetes Dashboard、监控系统、日志系统等必要的附加组件则不在其考虑范围之内，需要管理员按需自行部署。

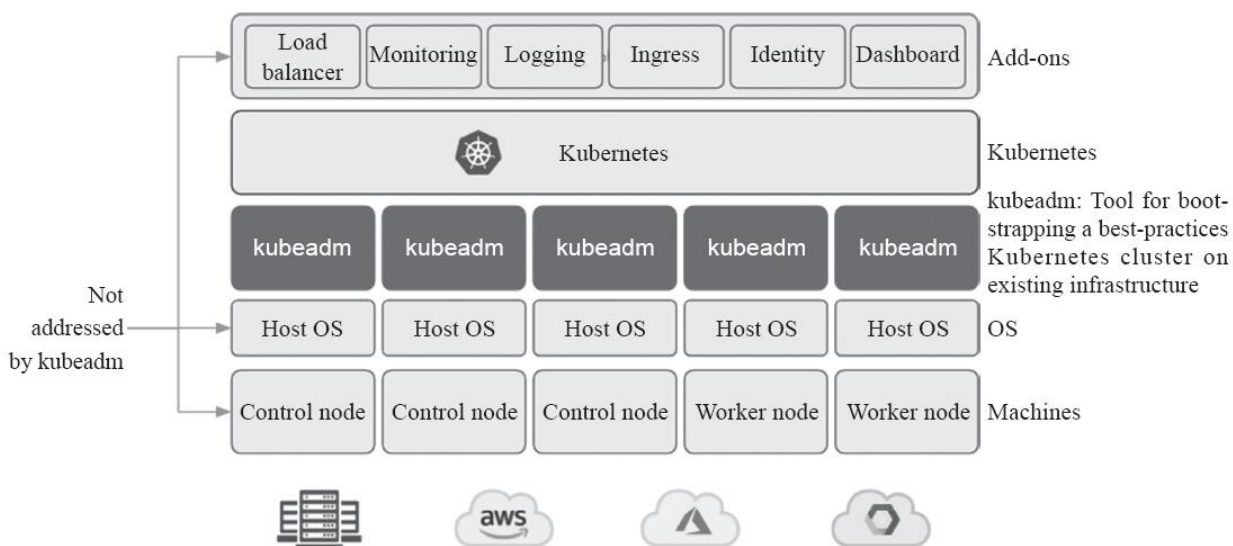


图2-6 kubeadm功能示意图

kubeadm集成了kubeadm init和kubeadm join等工具程序，其中kubeadm init用于集群的快速初始化，其核心功能是部署Master节点的各个组件，而kubeadm join则用于将节点快速加入到指定集群中，它们是创建Kubernetes集群最佳实践的“快速路径”。另外，kubeadm token可用于集群构建后管理用于加入集群时使用的认证令牌（token），而kubeadm reset命令的功能则是删除集群构建过程中生成的文件以重置回初始状态。

kubeadm还支持管理初始引导认证令牌（Bootstrap Token），完成待加入的新节点首次联系API Server时的身份认证（基于共享密钥）。另外，它们还支持管理集群版本的升级和降级操作。Kubernetes 1.8版

本之前，**kubeadm**一直处于**beta**级别，并警告不能用于生产环境。不过，自1.9版本开始，其虽仍处于**beta**版本，但已经不再输出警告信息，而随着1.11版本发布的**kubeadm**又得到了进一步的增强，它支持动态配置**kubelet**，通过增强的**CRI**集成支持动态探测以判定所用的容器引擎，并引入了几个新的命令行工具，包括**kubeadm config print-default**、**kubeadm config migrate**、**kubeadm config images pull**和**kubeadm upgrade node config**等。总体来说，使用**kubeadm**部署Kubernetes集群具有如下几个方面的优势。

- 简单易用：**kubeadm**可完成集群的部署、升级和拆除操作，并且对新手用户非常友好。

- 适用领域广泛：支持将集群部署于裸机、VMware、AWS、Azure、GCE及更多环境的主机上，且部署过程基本一致。

- 富有弹性：1.11版中的**kubeadm**支持阶段式部署，管理员可分为多个独立步骤完成部署操作。

- 生产环境可用：**kubeadm**遵循以最佳实践的方式部署Kubernetes集群，它强制启用**RBAC**，设定Master的各组件间以及API Server与kublet之间进行认证及安全通信，并锁定了**kubelet API**等。

由此可见，**kubeadm**并非一键安装类的解决方案，相反，它有着更宏大的目标，旨在成为一个更大解决方案的一部分，试图为集群创建和运营构建一个声明式的API驱动模型，它将集群本身视为不可变组件，而升级操作等同于全新部署或就地更新。目前，使用**kubeadm**部署集群已经成为越来越多的Kubernetes工程师的选择。

2.2.2 集群运行模式

Kubernetes集群支持三种运行模式：一是“独立组件”模式，系统各组件直接以守护进程的方式运行于节点之上，各组件之间相互协作构成集群，如图2-7b所示；第二种是“静态Pod模式”，除kubelet和Docker之外的其他组件（如etcd、kube-apiserver、kube-controller-manager和kube-scheduler等）都是以静态Pod对象运行于Master主机之上的，如图2-7a所示；第三种是Kubernetes的“自托管”（self-hosted）模式，它类似于第二种方式，将除了kubelet和Docker之外的其他组件运行在集群之上的Pod对象，但不同的是，这些Pod对象托管运行在集群自身之上受控于DaemonSet类型的控制器，而非静态的Pod对象。

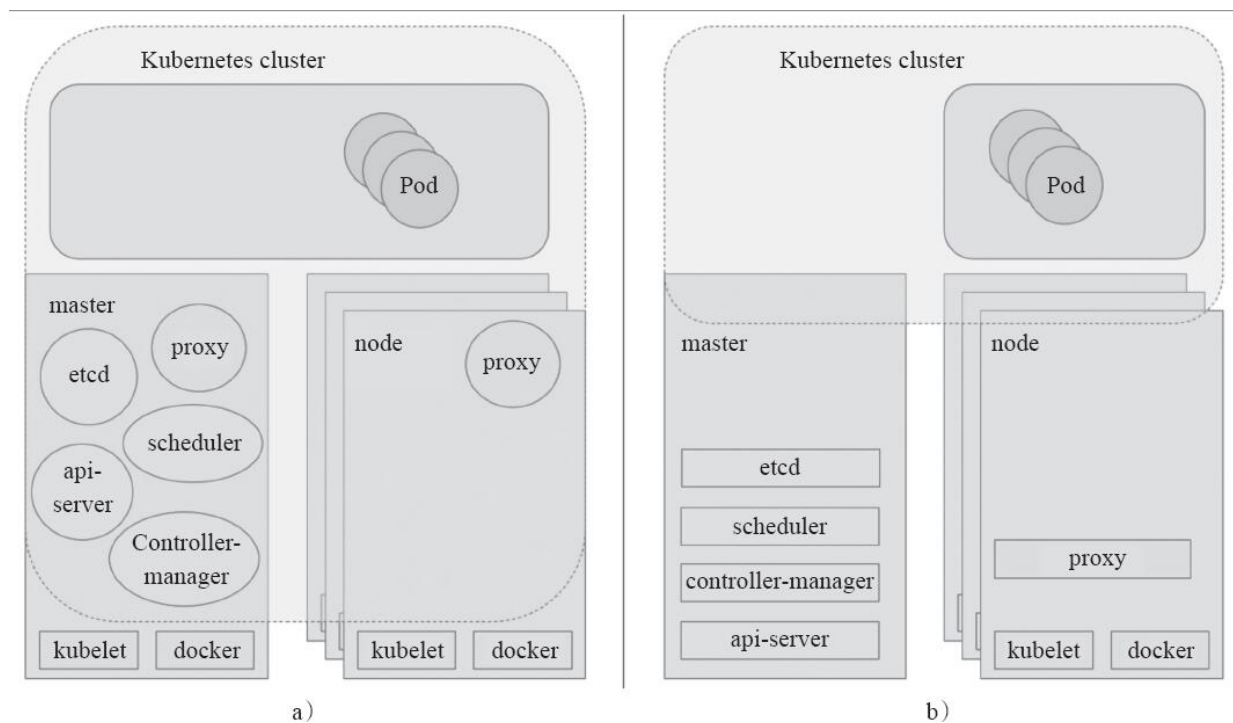


图2-7 Kubernetes集群的运行模式（图2-7a为静态Pod模式）

使用kubeadm部署的Kubernetes集群可运行第二种或第三种模式，默认为静态Pod对象模式，需要使用自托管模式时，kubeadm init命令使用“--features-gates=selfHosting”选项即可。第一种模式集群的构建需要将各组件运行于系统之上的独立守护进程中，其间需要用到的证书及Token等认证信息也都需要手动生成，过程烦琐且极易出错；若有

必要用到，则建议使用GitHub上合用的项目辅助进行，例如，通过ansible playbook进行自动部署等。

2.2.3 准备用于实践操作的集群环境

本书后面的篇幅中用到的测试集群如图2-8所示，该集群由一个Master主机和三个Node主机组成，它基于kubeadm部署，除了kubelet和Docker之外其他的集群组件都运行于Pod对象中。多数情况下，两个或以上的独立运行的Node主机即可测试分布式集群的核心功能，因此其数量可按需定义，但两个主机是模拟分布式环境的最低需求。生产实践中，应该至少部署三个协同工作的Master节点以确保控制平面的服务可用性，不过，在测试环境中仅部署一个Master节点也是常见的选择。

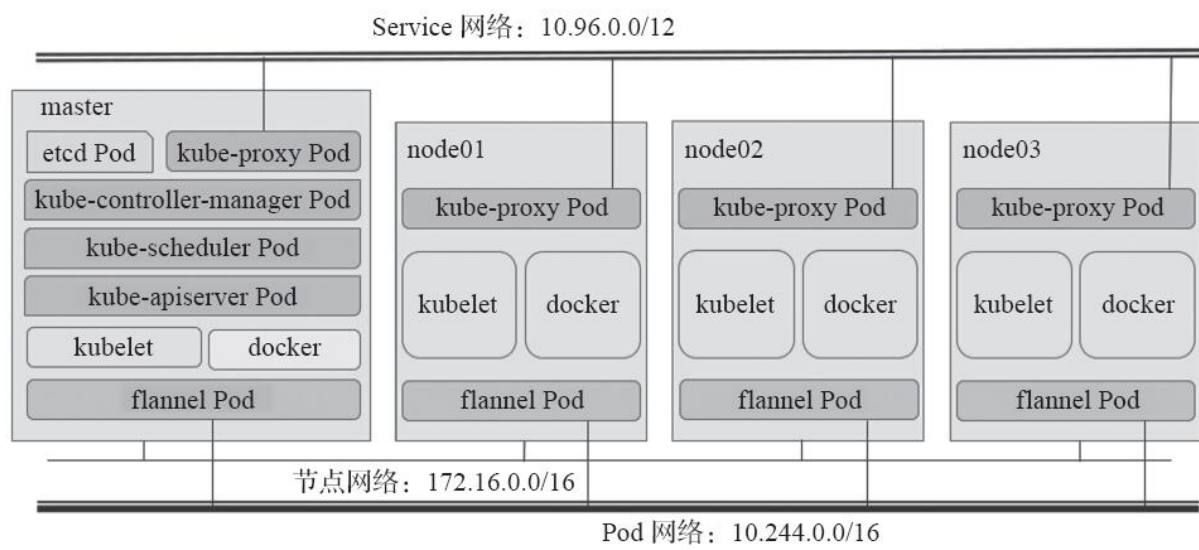


图2-8 Kubernetes集群部署目标示意图

各Node上采用的容器运行时环境为docker，后续的众多容器的运行任务都将依赖于Docker Registry服务，包括DockerHub、GCR（Google Container Registry）和Quay等，甚至是私有的Registry服务，本书假设读者对Docker容器技术有熟练的使用基础。另外，本部署示例中使用的用于为Pod对象提供网络功能的插件是flannel，其同样以Pod对象的形式托管运行于Kubernetes系统之上。

具体的部署过程以及本书用到的集群环境请参考附录A，本章后续的操作都将依赖于根据其步骤部署完成的集群环境，读者需要根据其内容成功搭建出Kubernetes测试集群后才能进行后面章节的学习。

2.2.4 获取集群环境相关的信息

Kubernetes系统目前仍处于快速迭代阶段，版本演进频繁，读者所部署的版本与本书中使用的版本或将有所不同，其功能特性也将存在一定程度的变动。因此，事先查看系统版本，以及对比了解不同版本间的功能特性变动也将是不可或缺的步骤。当然，用户也可选择安装与本书相同的系统版本。下面的命令显示的是当前使用的客户端及服务端程序版本信息：

```
[root@master ~]# kubectl version --short=true
Client Version: v1.12.1
Server Version: v1.12.1
```



提示 Kubernetes系统版本变动时的ChangeLog可参考github.com站点上相关版本中的介绍。

Kubernetes集群以及部署的附件CoreDNS等提供了多种不同的服务，客户端访问这些服务时需要事先了解其访问接口，管理员可使用“`kubectl cluster-info`”命令获取相关的信息。

```
[root@master ~]# kubectl cluster-info
Kubernetes master is running at https://172.16.0.70:6443
CoreDNS is running at https://172.16.0.70:6443/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy
```

一个功能完整的Kubernetes集群应当具备的附加组件还包括Dashboard、Ingress Controller和Heapster（或Prometheus）等，后续章节中的某些概念将会依赖到这些组件，读者可选择在将要用到时再进行部署。

2.3 kubectl使用基础与示例

Kubernetes API是管理其各种资源对象的唯一入口，它提供了一个RESTful风格的CRUD（Create、Read、Update和Delete）接口用于查询和修改集群状态，并将结果存储于集群状态存储系统etcd中。事实上，API server也是用于更新etcd中资源对象状态的唯一途径，Kubernetes的其他所有组件和客户端都要通过它来完成查询或修改操作，如图2-9所示。从这个角度来讲，它们都算得上是API server的客户端。

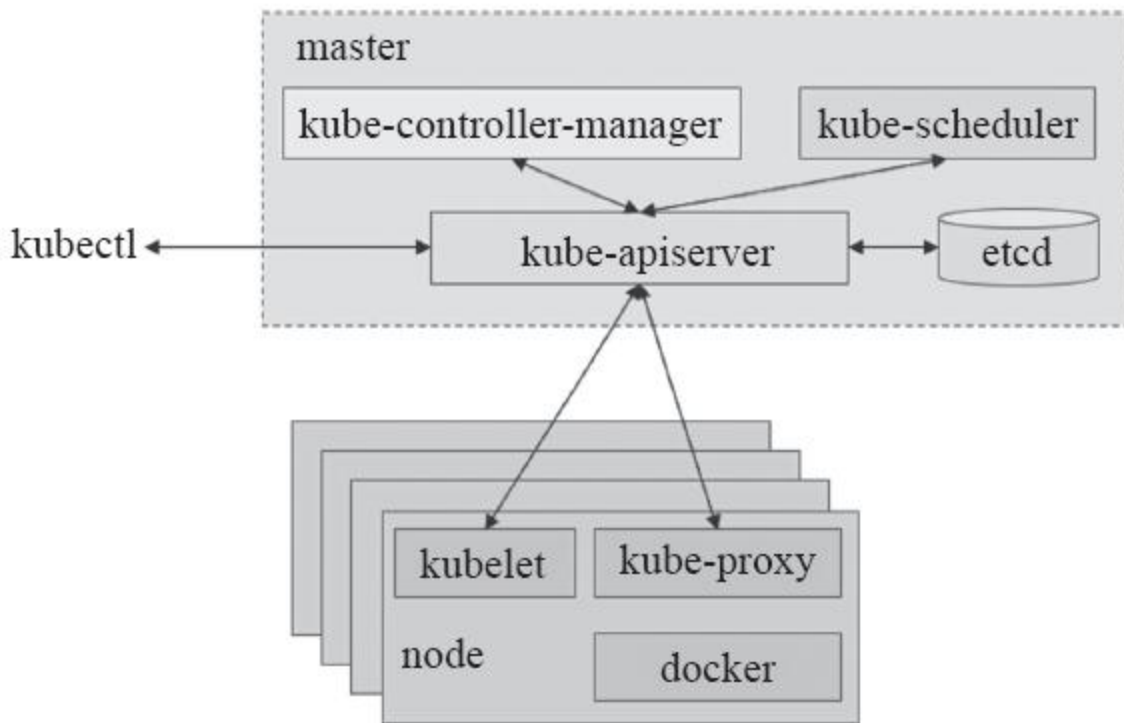


图2-9 API Server及其部分客户端

任何RESTful风格API中的核心概念都是“资源”（resource），它是具有类型、关联数据、同其他资源的关系以及可对其执行的一组操作方法的对象，它与对象式编程语言中的对象实例类似，两者之间的重要区别在于RESTful API仅为资源定义了少量的标准方法（对应于标准HTTP的GET、POST、PUT和DELETE方法），而编程语言中的对象实例通常有很多方法。另外，资源可以根据其特性分组，每个组是同一类型资源的集合，它仅包含一种类型的资源，并且各资源间不存在

顺序的概念，集合本身也是资源。对应于Kubernetes中，Pod、Deployment和Service等都是所谓的资源类型，它们由相应类型的对象集合而成。

API Server通过认证（Authentication）、授权（Authorization）和准入控制（Admission Control）等来管理对资源的访问请求，因此，来自于任何客户端（如kubectl、kubelet、kube-proxy等）的访问请求都必须事先完成认证之后方可进行后面的其他操作。API Server支持多种认证方式，客户端可以使用命令行选项或专用的配置文件（称为kubeconfig）提供认证信息。相关的内容将在后面的章节中给予详细说明。

kubectl的核心功能在于通过API Server操作Kubernetes的各种资源对象，它支持三种操作方式，其中直接命令式（Imperative commands）的使用最为简便，是了解Kubernetes集群管理的一种有效途径。

kubectl命令常用操作示例

为了便于读者快速适应kubectl的命令操作，这里给出几个使用示例用于说明其基本使用方法。

1.创建资源对象

直接通过kubectl命令及相关的选项创建资源对象的方式即为直接命令式操作，例如下面的命令分别创建了名为nginx-deploy的Deployment控制器资源对象，以及名为nginx-svc的Service资源对象：

```
$ kubectl run nginx-deploy --image=nginx:1.12 --replicas=2
$ kubectl expose deployment/nginx --name=nginx-svc --port=80
```

用户也可以根据资源清单创建资源对象，即命令式对象配置文件，例如，假设存在定义了Deployment对象的nginx-deploy.yaml文件，和定义了Service对象的nginx-svc.yaml文件，使用kubectl create命令即可进行基于命令式对象配置文件的创建操作：

```
$ kubectl create -f nginx-deploy.yaml -f nginx-svc.yaml
```

甚至还可以将创建交由**kubectl**自行确定，用户只需要声明期望的状态，这种方式称为声明式对象配置。例如，假设存在定义了**Deployment**对象的**nginx-deploy.yaml**文件，以及定义了**Service**对象的**nginx-svc.yaml**文件，那么使用**kubectl apply**命令即可实现声明式配置：

```
$ kubectl apply -f nginx-deploy.yaml -f nginx-svc.yaml
```

本章后面的章节主要使用第一种资源管理方式，第二种和第三种方式将在后面的章节中展开讲述。

2.查看资源对象

运行着实际负载的**Kubernetes**系统上通常会存在多种资源对象，用户可分类列出感兴趣的资源对象及其相关的状态信息，“**kubectl get**”正是用于完成此类功能的命令。例如，列出系统上所有的**Namespace**资源对象，命令如下：

```
$ kubectl get namespaces
```

用户也可一次查看多个资源类别下的资源对象，例如，列出默认名称空间内的所有**Pod**和**Service**对象，并输出额外信息，可以使用如下形式的**kubectl get**命令：

```
$ kubectl get pods, services -o wide
```

Kubernetes系统的大部分资源都隶属于某个**Namespace**对象，缺省的名称空间为**default**，若需要获取指定**Namespace**对象中的资源对象的信息，则需要使用**-n**或**--namespace**指明其名称。例如，列出**kube-namespace**名称空间中拥有**k8s-app**标签名称的所有**Pod**对象：

```
$ kubectl get pods -l k8s-app -n kube-system
```

3.打印资源对象的详细信息

每个资源对象都包含着用户期望的状态（Spec）和现有的实际状态（Status）两种状态信息，“`kubectl get -o {yaml|json}`”或“`kubectl describe`”命令都能够打印出指定资源对象的详细描述信息。例如，查看kube-system名称空间中拥有标签component=kube-apiserver的Pod对象的资源配置清单（期望的状态）及当前的状态信息，并输出为yaml格式，命令如下：

```
$kubectl get pods -l component=kube-apiserver -o yaml -n kube-system
```

而“`kubectl describe`”命令还能显示与当前对象相关的其他资源对象，如Event或Controller等。例如，查看kube-system名称空间中拥有标签component=kube-apiserver的Pod对象的详细描述信息，可以使用下面的命令：

```
$kubectl describe pods -l component=kube-apiserver -n kube-system
```

这两个命令都支持以“TYPE NAME”或“TYPE/NAME”的格式指定具体的资源对象，如“`pods kube-apiserver-master.ilinux.io`”或“`pods/kube-apiserver-master.ilinux.io`”，以了解特定资源对象的详细属性信息及状态信息。

4.打印容器中的日志信息

通常一个容器中仅会运行一个进程（及其子进程），此进程作为PID为1的进程接收并处理管理信息，同时将日志直接输出至终端中，而无须再像传统的多进程系统环境那样将日志保存于文件中，因此容器日志信息的获取一般要到其控制上进行。“`kubectl logs`”命令可打印Pod对象内指定容器的日志信息，命令格式为“`kubectl logs [-f] [-p]`

（`POD|TYPE/NAME`）`[-c CONTAINER][options]`”，若Pod对象内仅有一个容器，则-c选项及容器名为可选。例如，查看名称空间kube-system中仅有一个容器的Pod对象kube-apiserver-master.ilinux.io的日志：

```
$kubectl logs kube-apiserver-master.ilinux.io -n kube-system
```

为上面的命令添加“-f”选项，还能用于持续监控指定容器中的日志输出，其行为类似于使用了-f选项的tail命令。

5.在容器中执行命令

容器的隔离属性使得对其内部信息的获取变得不再直观，这一点在用户需要了解容器内进程的运行特性、文件系统上的文件及路径布局等信息时，需要穿透其隔离边界进行。“kubectl exec”命令便是用于在指定的容器内运行其他应用程序的命令，例如，在kube-system名称空间中的Pod对象kube-apiserver-master.ilinux.io上的唯一容器中运行ps命令：

```
$kubectl exec kube-apiserver-master.ilinux.io -n kube-system -- ps
```

注意，若Pod对象中存在多个容器，则需要以-c选项指定容器后再运行。

6.删除资源对象

使命已经完成或存在错误的资源对象可使用“kubectl delete”命令予以删除，不过，对于受控于控制器的对象来说，删除之后其控制器可能会重建出类似的对象，例如，Deployment控制器下的Pod对象在被删除时就会被重建。例如，删除默认名称空间中名为nginx-svc的Service资源对象：

```
$kubectl delete services nginx-svc
```

下面的命令可用于删除kube-system名称空间中拥有标签“k8s-app=kube-proxy”的所有Pod对象：

```
$kubectl delete pods -l app=monitor -n kube-system
```

若要删除指定名称空间中的所有的某类对象，可以使用“kubectl delete TYPE--all-n NS”命令，例如，删除kube-public名称空间中的所有

Pod对象:

```
$kubectl delete pods--all-n kube-public
```

另外，有些资源类型（如**Pod**），支持优雅删除的机制，它们有着默认的删除宽限期，不过，用户可以在命令中使用**--grace-period**选项或**--now**选项来覆盖默认的宽限期。

2.4 命令式容器应用编排

本节将使用示例镜像“`ikubernetes/myapp: v1`”来演示容器应用编排的基础操作：应用部署、访问、查看、服务暴露和扩缩容等。一般说来，Kubernetes之上应用程序的基础管理操作由如下几个部分组成。

- 1) 通过合用的Controller类的资源（如Deployment或ReplicationController）创建并管控Pod对象以运行特定的应用程序，如Nginx或tomcat等。无状态（stateless）应用的部署和控制通常使用Deployment控制器进行，而有状态应用则需要使用StatefulSet控制器。
- 2) 为Pod对象创建Service对象，以便向客户端提供固定的访问路径，并借助于CoreDNS进行服务发现。
- 3) 随时按需获取各资源对象的简要或详细信息，以了解其运行状态。
- 4) 如有需要，则手动对支持扩缩容的Controller组件进行扩容或缩容；或者，为支持HPA的Controller组件（如Deployment或ReplicationController）创建HPA资源对象以实现Pod副本数目的自动伸缩。
- 5) 滚动更新：当应用程序的镜像出现新版本时，对其执行更新操作；必要时，为Pod对象中的容器更新其镜像版本；并可根据需要执行回滚操作。

本节中的操作示例仅演示了前三个部分的功能，即应用的部署、服务暴露及相关信息的查看。应用的扩缩容、升级及回滚等操作会在后面的章节中进行详细介绍。



提示 以下操作命令在任何部署了kubectl并能正常访问到Kubernetes集群的主机上均可执行，包括集群外的主机。复制master主机上的`/etc/kubernetes/admin.conf`至相关用户主目录下的`.kube/config`文件即可正常执行，具体方法请参考`kubeadm init`命令结果中的提示。

2.4.1 部署应用（Pod）

在Kubernetes集群上自主运行的Pod对象在非计划内终止后，其生命周期即告结束，用户需要再次手动创建类似的Pod对象才能确保其容器中的应用依然可得。对于Pod数量众多的场景，尤其是对微服务业务来说，用户必将疲于应付此类需求。Kubernetes的工作负载

（workload）类型的控制器能够自动确保由其管控的Pod对象按用户期望的方式运行，因此，Pod的创建和管理大多都会通过这种类型的控制器来进行，包括Deployment、ReplicaSet、ReplicationController等。

1. 创建Deployment控制器对象

“kubectl run”命令可于命令行直接创建Deployment控制器，并以--image选项指定的镜像运行Pod中的容器，--dry-run选项可用于命令的测试运行，但并未真正执行资源对象的创建过程。例如，下面的命令要创建一个名为myapp的Deployment控制器对象，它使用镜像ikubernetes/myapp: v1创建Pod对象，但仅在测试运行后即退出：

```
-]$ kubectl run myapp --image=ikubernetes/myapp:v1 --port=80 --replicas=1
--dry-run
NAME          AGE
myapp         <unknown>
```

镜像ikubernetes/myapp: v1中定义的容器主进程为默认监听于80端口的Web服务程序Nginx，因此，如下命令使用“--port=80”来指明容器要暴露的端口。而“--replicas=1”选项则指定了目标控制器对象要自动创建的Pod对象的副本数量。确认测试命令无误后，可移除“--dry-run”选项后再次执行命令以完成资源对象的创建：

```
-]$ kubectl run myapp --image=ikubernetes/myapp:v1 --port=80 --replicas=1
deployment.apps/myapp created
```

创建完成后，其运行效果示意图如图2-10所示，它在default名称空间中创建了一个名为myapp的Deployment控制器对象，并由它基于指定的镜像文件创建了一个Pod对象。

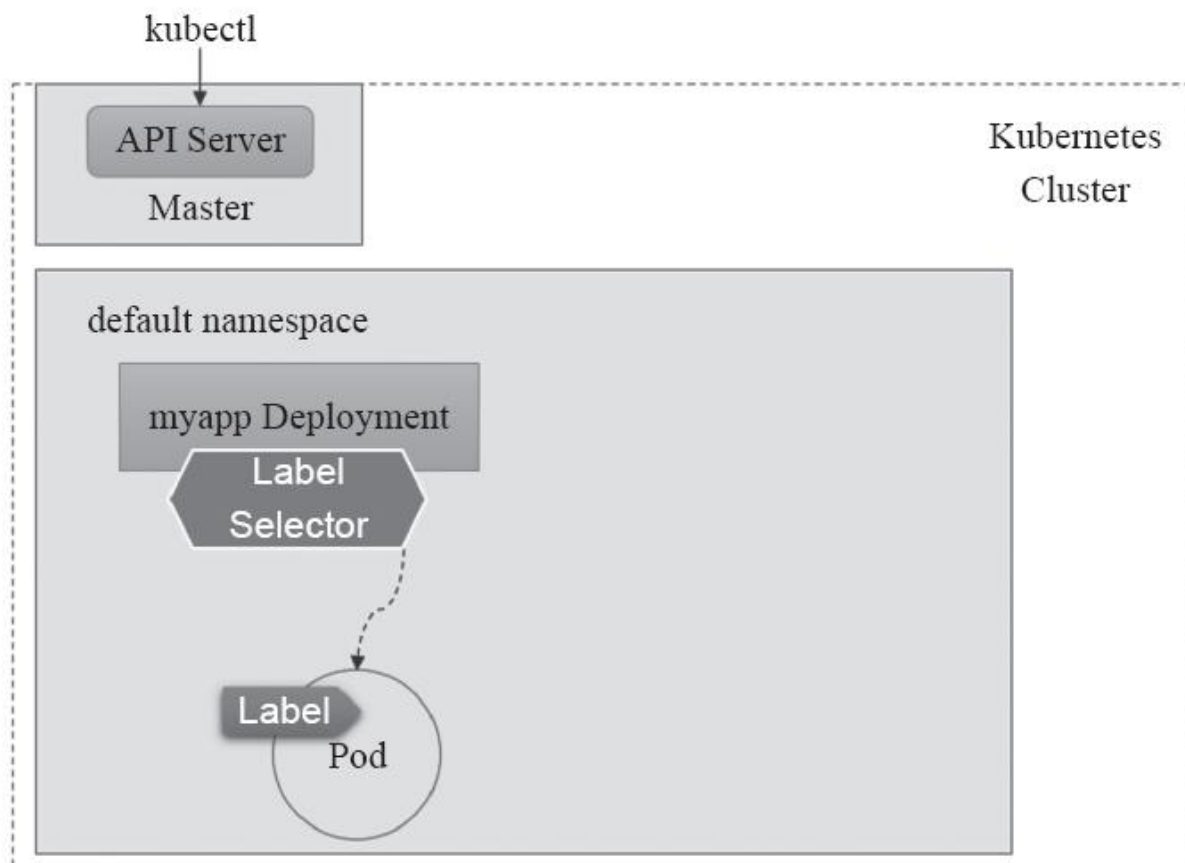


图2-10 Deployment对象myapp及其创建的Pod对象

`kubectl run`命令其他常用的选项还有如下几个，它们支持用户在创建资源对象时实现更多的控制，具体如下。

- `-l`, `--labels`: 为Pod对象设定自定义标签。
- `--record`: 是否将当前的对象创建命令保存至对象的Annotation中，布尔型数据，其值可为true或false。
- `--save-config`: 是否将当前对象的配置信息保存至Annotation中，布尔型数据，其值可为true或false。
- `--restart=Never`: 创建不受控制器管控的自主式Pod对象。

其他可用选项及使用方式可通过“`kubectl run--help`”命令获取。资源对象创建完成后，通常需要了解其当前状态是否正常，以及是否能够

吻合于用户期望的目标状态，相关的操作一般使用`kubectl get`、`kubectl describe`等命令进行。

2.打印资源对象的相关信息

`kubectl get`命令可用于获取各种资源对象的相关信息，它既能够显示对象类型特有格式的简要信息，也能够指定出格式为YAML或JSON的详细信息，或者使用Go模板自定义要显示的属性及信息等。例如，下面是查看前面创建的Deployment对象的相关运行状态的命令及其输出结果：

~]\$ kubectl get deployments					
NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
myapp	1	1	1	1	1m

上面命令的执行结果中，各字段的说明具体如下。

- 1) **NAME**: 资源对象的名称。
- 2) **DESIRED**: 用户期望由当前控制器管理的Pod对象副本的精确数量。
- 3) **CURRENT**: 当前控制器已有的Pod对象的副本数量。
- 4) **UP-TO-DATE**: 更新到最新版本定义的Pod对象的副本数量，在控制器的滚动更新模式下，它表示已经完成版本更新的Pod对象的副本数量。
- 5) **AVAILABLE**: 当前处于可用状态的Pod对象的副本数量，即可正常提供服务的副本数。
- 6) **AGE**: Pod的存在时长。



提示 Deployment资源对象通过ReplicaSet控制器实例完成对Pod对象的控制，而非直接控制。另外，通过控制器创建的Pod对象都会被自动附加一个标签，其格式为“run=<Controller_Name>”，例如，上面的命令所创建的Pod，会拥有“run=myapp”标签。后面的章节对此会有详细描述。

而此Deployment控制器创建的唯一Pod对象运行正常与否，其被调度至哪个节点运行，当前是否就绪等也是用户在创建完成后应该重点关注的信息。由控制器创建的Pod对象的名称通常是以控制器名称为前缀，以随机字符为后缀，例如，下面命令输出结果中的myapp-6865459dff-5nsjc:

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myapp-6865459dff-5nsjc ilinux.io	1/1	Running	0	3m	10.244.3.2	node03.

上面命令的执行结果中，每一个字段均代表着Pod资源对象一个方面的属性，除了NAME之外的其他字段及功用说明如下。

①READY: Pod中的容器进程初始化完成并能够正常提供服务时即为就绪状态，此字段用于记录处于就绪状态的容器数量。

②STATUS: Pod的当前状态，其值可能是Pending、Running、Succeeded、Failed和Unknown等其中之一。

③RESTARTS: Pod对象可能会因容器进程崩溃、超出资源限额等原因发生故障问题而被重启，此字段记录了它重启的次数。

④IP: Pod的IP地址，其通常由网络插件自动分配。

⑤NODE: 创建时，Pod对象会由调度器调度至集群中的某节点运行，此字段即为节点的相关标识信息。



提示 如果指定名称空间中存在大量的Pod对象而使得类似如上命令的输出结果存在太多的不相关信息时，则可通过指定选项“-l run=myapp”进行Pod对象过滤，其仅显示符合此标签选择器的Pod对象。

确认Pod对象已转为“Running”状态之后，即可于集群中的任一节点（或其他Pod对象）直接访问其容器化应用中的服务，如图2-11中节点NodeX上的客户端程序Client，或者集群上运行于Pod中的客户端程序。

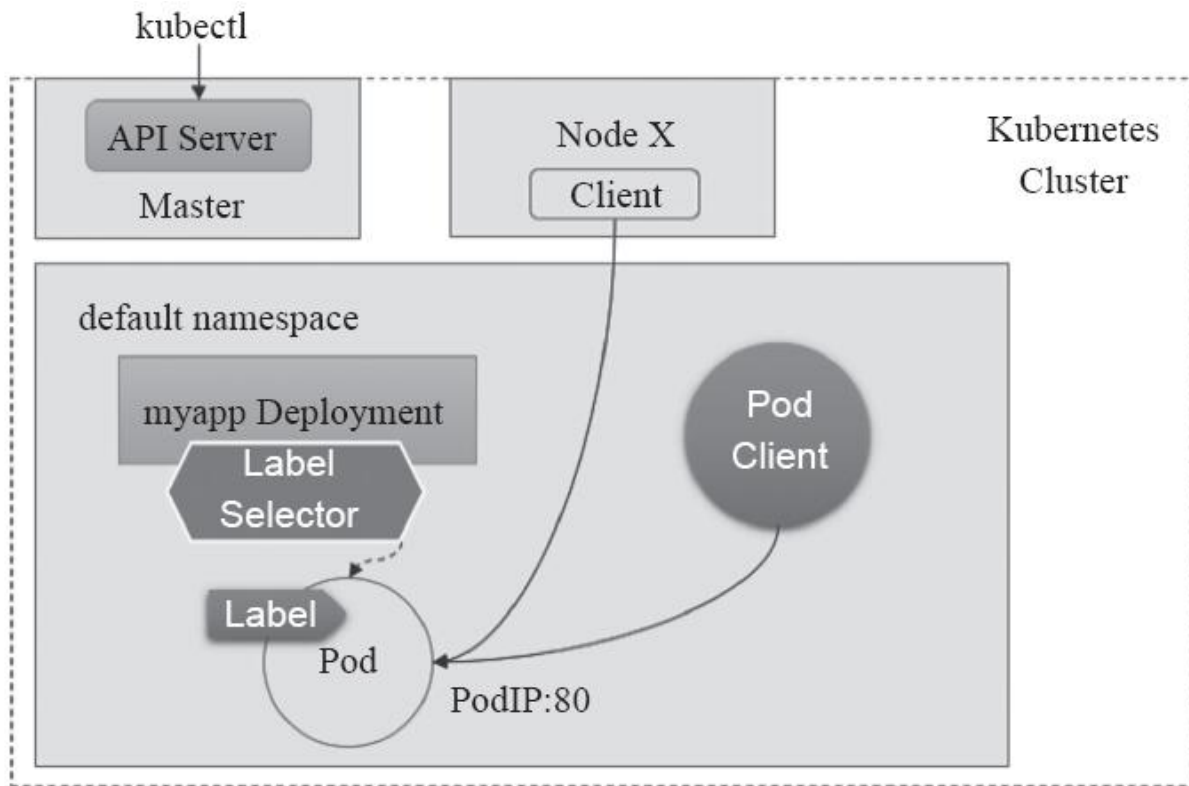


图2-11 访问Pod中容器化应用程序

例如，在集群中任一节点上使用curl命令对地址为10.244.3.2的Pod对象myapp-6865459dff-5nsjc的80端口发起服务请求，命令及结果如下所示：

```
[ik8s@node03 ~]$ curl http://10.244.3.2:80/  
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

2.4.2 探查Pod及应用详情

资源创建或运行过程中偶尔会因故出现异常，此时用户需要充分获取相关的状态及配置信息以便确定问题的所在。另外，在对资源对象进行创建或修改完成之后，也需要通过其详细的状态来了解操作成功与否。**kubectl**有多个子命令可用于从不同的角度显示对象的状态信息，这些信息有助于用户了解对象的运行状态、属性详情等信息。

1) **kubectl describe**: 显示资源的详情，包括运行状态、事件等信息，但不同的资源类型其输出内容不尽相同。

2) **kubectl logs**: 查看Pod对象中容器输出在控制台的日志信息。在Pod中运行有多个容器时，需要使用选项“-c”指定容器名称。

3) **kubectl exec**: 在Pod对象某容器内运行指定的程序，其功能类似于“**docker exec**”命令，可用于了解容器各方面的相关信息或执行必需的设定操作等，其具体功能取决于容器内可用的程序。

1.查看Pod对象的详细描述

下面给出的命令打印了此前由**myapp**创建的Pod对象的详细状态信息，为了便于后续的多次引用，这里先将其名称保存于变量**POD_NAME**中。命令的执行结果中省略了部分输出：

```
~]$ POD_NAME=myapp-6865459dff-5nsjc
~]$ kubectl describe pods $POD_NAME
Name:                myapp-6865459dff-5nsjc
Namespace:           default
Priority:             0
PriorityClassName:    <none>
Node:                node03.ilinux.io/172.16.0.68
.....
Status:              Running
IP:                  10.244.3.2
Controlled By:       ReplicaSet/myapp-6865459dff
Containers:
  myapp:
    .....
.....
Events:
  Type      Reason      Age   From                  Message
  ---      -
  Normal    Scheduled   55m   default-scheduler     Successfully assigned
```

```
default/myapp-6865459dff-5nsjc to node03.ilinux.io
Normal Pulling      55m   kubelet, node03.ilinux.io   pulling image "ikubernetes/
myapp:v1"
Normal Pulled       54m   kubelet, node03.ilinux.io   Successfully pulled image
"ikubernetes/myapp:v1"
Normal Created      54m   kubelet, node03.ilinux.io   Created container
Normal Started      54m   kubelet, node03.ilinux.io   Started container
```

不同的需求场景中，用户需要关注不同纬度的输出，但一般来说，**Events**和**Status**字段会是重点关注的对象，它们分别代表了**Pod**对象运行过程中的重要信息及当前状态。上面命令执行结果中的不少字段都可以见名而知义，而且部分字段在前面介绍其他命令输出时已经给出，还有一部分会在本书后面的篇幅中给予介绍。

2.查看容器日志

Docker容器一般仅运行单个应用程序，其日志信息将通过标准错误输出等方式直接打印至控制台，“**kubectl logs**”命令即用于查看这些日志。例如，查看由**Deployment**控制器**myapp**创建的**Pod**对象的控制台日志，命令如下：

```
~]$ kubectl logs $POD_NAME
10.244.3.1 - - [.....] "GET / HTTP/1.1" 200 65 "-" "curl/7.29.0" "-"
.....
```

如果**Pod**中运行有多个容器，则需要在查看日志时为其使用“-c”选项指定容器名称。例如，当读者所部署的是**KubeDNS**附件而非**CoreDNS**时，**kube-system**名称空间内的**kube-dns**相关的**Pod**中同时运行着**kubedns**、**dnsmasq**和**sidecar**三个容器，如果要查看**kubedns**容器的日志，需要使用类似如下的命令：

```
~]$ DNS_POD=$(kubectl get pods -o name -n kube-system | grep kube-dns)
~]$ kubectl logs $DNS_POD -c kubedns -n kube-system
```

需要注意的是，日志查看命令仅能用于打印存在于**Kubernetes**系统上的**Pod**中容器的日志，对于已经删除的**Pod**对象，其容器日志信息将无从获取。日志信息是用于辅助用户获取容器中应用程序运行状态的最有效的途径之一，也是非常重要的排错手段，因此通常需要使用集中式的日志服务器统一收集存储于各**Pod**对象中容器的日志信息。

3.在容器中运行额外的程序

运行着非交互式进程的容器中，默认运行的唯一进程及其子进程启动后，容器即进入独立、隔离的运行状态。对容器内各种详情的了解需要穿透容器边界进入其中运行其他的应用程序来进行，“**kubectl exec**”可以让用户在Pod的某容器中运行用户所需要的任何存在于容器中的程序。在“**kubectl logs**”获取的信息不够全面时，此命令可以通过在Pod中运行其他指定的命令（前提是容器中存在此程序）来辅助用户获取更多的信息。一个更便捷的使用接口的方式是直接交互式运行容器中的某个Shell程序。例如，直接查看Pod中的容器运行的进程：

```
~]$ kubectl exec $POD_NAME ps aux
PID    USER      TIME    COMMAND
  1  root          0:00  nginx: master process nginx -g daemon off;
  8  nginx       0:00  nginx: worker process
  9  root          0:00  ps aux
```



注意 如果Pod对象中运行了多个容器，那么在程序运行时还需要使用“-c<container_name>”选项指定要于其内部运行程序的容器名称。

若要进入容器的交互式Shell接口，可使用类似如下的命令，斜体部分表示在容器的交互式接口中执行的命令：

```
~]$ kubectl -it exec $POD_NAME /bin/sh
/ # hostname
myapp-6865459dff-5nsjc
/ # netstat -tnl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80               0.0.0.0:*               LISTEN
/ #
```

2.4.3 部署Service对象

简单来说，一个Service对象可视作通过其标签选择器过滤出的一组Pod对象，并能够为此组Pod对象监听的套接字提供端口代理及调度服务。

1. 创建Service对象

“`kubectl expose`”命令可用于创建Service对象以将应用程序“暴露”（`expose`）于网络中。例如，下面的命令即可将myapp创建的Pod对象使用“NodePort”类型的服务暴露到集群外部：

```
~]$ kubectl expose deployments/myapp --type="NodePort" --port=80 --name=myapp
service "myapp" exposed
```

上面的命令中，`--type`选项用于指定Service的类型，而`--port`则用于指定要暴露的容器端口，目标Service对象的名称为myapp。创建完成后，default名称空间中的对象及其通信示意图如图2-12所示。

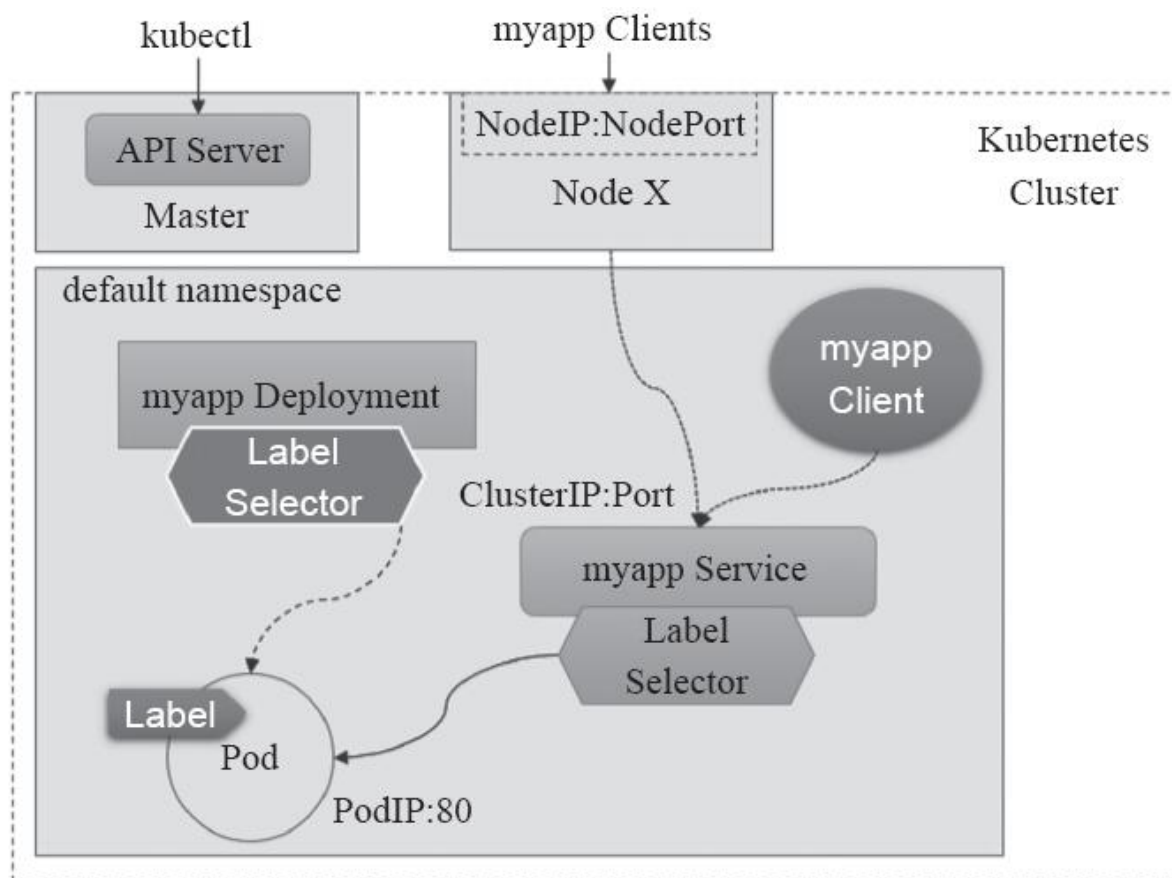


图2-12 Service对象在Pod对象前端添加了一个固定访问层

下面通过运行于同一集群中的Pod对象中的客户端程序发起访问测试，来模拟图2-12中的源自myapp Client Pod对象的访问请求。首先，使用kubectl run命令创建一个Pod对象，并直接接入其交互式接口，如下命令的-it组合选项即用于交互式打开并保持其shell命令行接口；而后通过wget命令对此前创建的Service对象的名称发起访问请求，如下命令中的myapp即Service对象名称，default即其所属的Namespace对象的名称：

```
$ kubectl run client --image=busybox --restart=Never -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # wget -O - -q http://myapp.default:80
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

创建时，Service对象名称及其ClusterIP会由CoreDNS附件动态添加至名称解析库当中，因此，名称解析服务在对象创建后即可直接使用。

类似于列出Deployment控制器及Pod对象的方式，“`kubectl get services`”命令能够列出Service对象的相关信息，例如下面的命令显示了Service对象myapp的简要状态信息：

~]\$ kubectl get svc/myapp					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp	NodePort	10.109.39.145	<none>	80:31715/TCP	5m

其中，“PORT (s)”字段表明，集群中各工作节点会捕获发往本地的目标端口为31715的流量，并将其代理至当前Service对象的80端口，于是，集群外部的用户可以使用当前集群中任一节点的此端口来请求Service对象上的服务。CLUSTER-IP字段为当前Service的IP地址，它是一个虚拟IP，并没有配置于集群中的任何主机的任何接口之上，但每个node之上的kube-proxy都会为CLUSTER-IP所在的网络创建用于转发的iptables或ipvs规则。此时，用户可于集群外部任一浏览器请求集群任一节点的相关端口来进行访问测试。

创建Service对象的另一种方式是使用“`kubectl create service`”命令，对应于每个类型，它分别有一个专用的子命令，例如“`kubectl create service clusterip`”和“`kubectl create service nodeport`”等，各命令在使用方式上也略有区别。

2.查看Service资源对象的描述

“`kubectl describe services`”命令用于打印Service对象的详细信息，它通常包括Service对象的Cluster IP，关联Pod对象时使用的标签选择器及关联到的Pod资源的端点等，示例如下：

~]\$ kubectl describe services myapp-svc	
Name:	myapp
Namespace:	default
Labels:	run=myapp
Annotations:	<none>
Selector:	run=myapp
Type:	NodePort
IP:	10.109.39.145
Port:	<unset> 80/TCP
TargetPort:	80/TCP
NodePort:	<unset> 31715/TCP
Endpoints:	10.244.3.2:80
Session Affinity:	None
External Traffic Policy:	Cluster
Events:	<none>

上面命令的执行结果输出基本上可以做到见名而知义，此处需要特别说明的几个字段具体如下：

1) **Selector**: 当前**Service**对象使用的标签选择器，用于选择关联的**Pod**对象。

2) **Type**: 即**Service**的类型，其值可以是**ClusterIP**、**NodePort**和**LoadBalancer**等其中之一。

3) **IP**: 当前**Service**对象的**ClusterIP**。

4) **Port**: 暴露的端口，即当前**Service**用于接收并响应请求的端口。

5) **TargetPort**: 容器中的用于暴露的目标端口，由**Service Port**路由请求至此端口。

6) **NodePort**: 当前**Service**的**NodePort**，它是否存在有效值与**Type**字段中的类型相关。

7) **EndPoints**: 后端端点，即被当前**Service**的**Selector**挑中的所有**Pod**的**IP**及其端口。

8) **Session Affinity**: 是否启用会话粘性。

9) **External Traffic Policy**: 外部流量的调度策略。

2.4.4 扩容和缩容

前面示例中创建的Deployment对象myapp仅创建了一个Pod对象，其所能承载的访问请求数量即受限于这单个Pod对象的服务容量。请求流量上升到接近或超出其容量之前，用户可以通过Kubernetes的“扩容机制”来扩展Pod的副本数量，从而提升其服务容量。

简单来说，所谓的“伸缩”（Scaling）就是指改变特定控制器上Pod副本数量的操作，“扩容”（scaling up）即为增加副本数量，而“缩容”（scaling down）则意指缩减副本数量。不过，无论是扩容还是缩容，其数量都需要由用户明确给出。

Service对象内建的负载均衡机制可在其后端副本数量不止一个时自动进行流量分发，它还会自动监控关联到的Pod的健康状态，以确保仅将请求流量分发至可用的后端Pod对象。若某Deployment控制器管理包含多个Pod实例，则必要时用户还可以为其使用“滚动更新”机制将其容器镜像升级到新的版本或变更那些支持动态修改的Pod属性。

使用kubectl run命令创建Deployment对象时，“--replicas=”选项能够指定由该对象创建或管理的Pod对象副本的数量，且其数量支持运行时进行修改，并立即生效。“kubectl scale”命令就是专用于变动控制器应用规模的命令，它支持对Deployment资源对象的扩容和缩容操作。例如，如果要将myapp的Pod副本数量扩展为3个，则可以使用如下命令来完成：

```
~]$ kubectl scale deployments/myapp --replicas=3
deployment.extensions "myapp" scaled
```

而后列出由myapp创建的Pod副本，确认其扩展操作的完成状态。如下命令显示出其Pod副本数量已经扩增至3个，其中包括此前的myapp-6865459dff-5nsjc：

```
~]$ kubectl get pods -l run=myapp
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-6865459dff-52j7t	0/1	ContainerCreating	0	53s
myapp-6865459dff-5nsjc	1/1	Running	0	2h
myapp-6865459dff-jz7t6	0/1	ContainerCreating	0	53s

Deployment对象myapp规模扩展完成之后，default名称空间中的资源对象及其关联关系如图2-13所示。

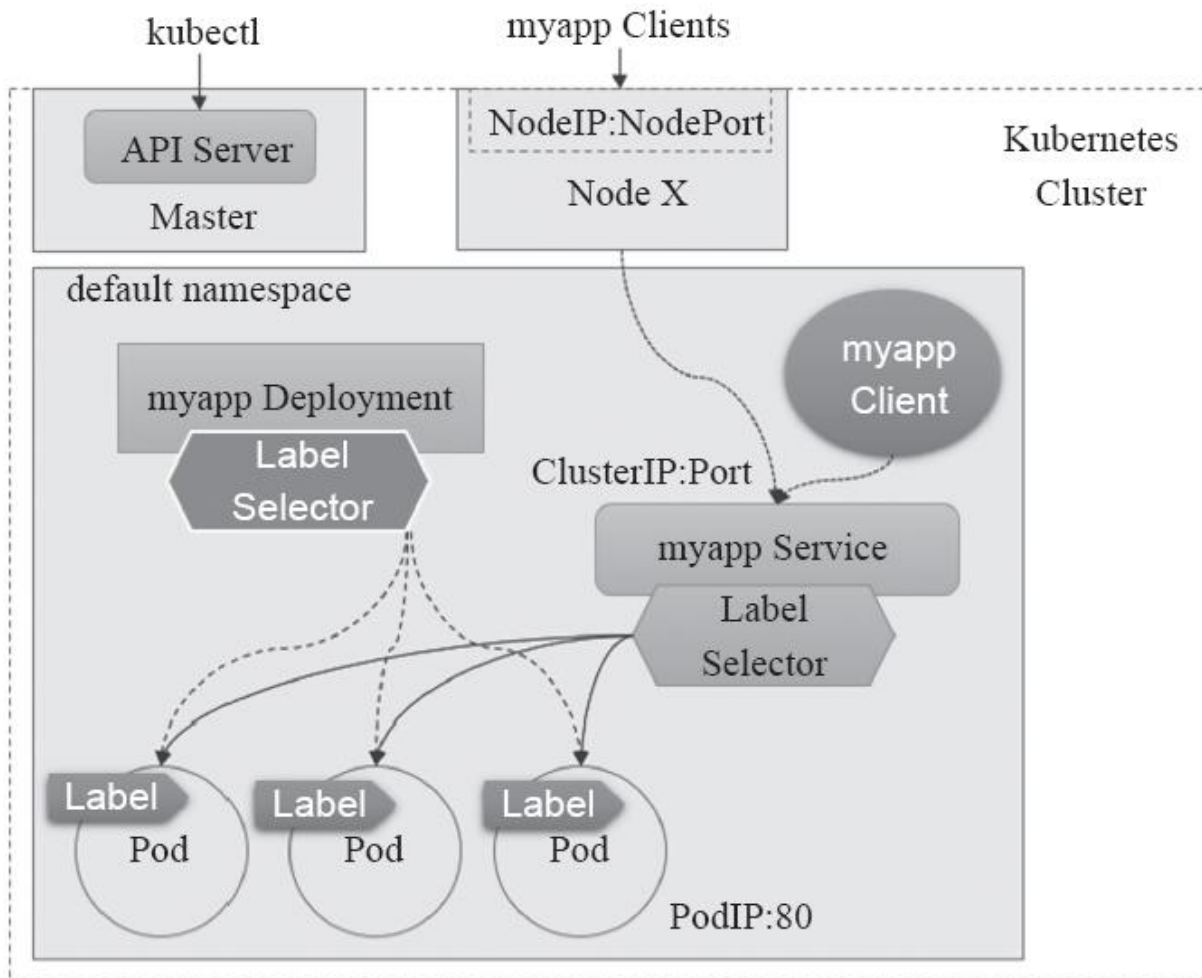


图2-13 Deployment对象规模扩增完成

而后由“`kubectl describe deployment`”命令打印Deployment对象myapp的详细信息，了解其应用规模的变动及当前Pod副本的状态等相关信息。从下面的命令结果可以看出，其Pod副本数量的各项指标都已经转换到了新的目标数量，而其事件信息中也有相应的事件显示其扩增操作已成功完成：

```
~]$ kubectl describe deployments/myapp
Name:                myapp
.....
Selector:            run=myapp
Replicas:            3 desired | 3 updated | 3 total | 3 available | 0
```

```
unavailable
.....
Events:
  Type          Reason          Age    From          Message
  ----          -
Normal ScalingReplicaSet 2m    deployment-controller Scaled up replica set myapp-6865459dff to 3
```

由myapp自动创建的Pod资源全都拥有同一个标签选择器“run=myapp”，因此，前面创建的Service资源对象myapp的后端端点也已经通过标签选择器自动扩展到了这3个Pod对象相关的端点，如下面的命令结果及图2-13所示：

```
~]$ kubectl describe services/myapp
Name:          myapp
.....
Endpoints:     10.244.1.3:80,10.244.2.2:80,10.244.3.2:80
.....
```

回到此前创建的客户端Pod对象client的交互式接口，对Service对象myapp反复发起测试请求，即可验证其负载均衡的效果。由如下命令及其结果可以看出，它会将请求调度至后端的各Pod对象进行处理：

```
~]$ / # while true; do wget -O - -q http://myapp.default:80/hostname.html;
    sleep 1; done
myapp-6865459dff-jz7t6
myapp-6865459dff-52j7t
myapp-6865459dff-5nsjc
.....
```

应用规模缩容的方式与扩容相似，只不过是Pod副本的数量调至比原来小的数字即可。例如，将myapp的Pod副本缩减至2个，可以使用如下命令进行：

```
~]$ kubectl scale deployments/myapp --replicas=2
deployment.extensions "myapp" scaled
```

至此，功能基本完整的容器化应用已在Kubernetes上部署完成，即便是一个略复杂的分层应用也只需要通过合适的镜像以类似的方式就能部署完成。

2.4.5 修改及删除对象

成功创建于Kubernetes之上的对象也称为活动对象（live object），其配置信息（live object configuration）由API Server保存于集群状态存储系统etcd中，“`kubectl get TYPE NAME-o yaml`”命令可获取到相关的完整信息，而运行“`kubectl edit`”命令可调用默认编辑器对活动对象的可配置属性进行编辑。例如，修改此前创建的Service对象myapp的类型为ClusterIP，使用“`kubectl edit service myapp`”命令打开编辑界面后修改type属性的值为ClusterIP，并删除NodePort属性，然后保存即可。对活动对象的修改将实时生效，但资源对象的有些属性并不支持运行时修改，此种情况下，编辑器将不允许保存退出。

有些命令是kubectl edit命令某一部分功能的二次封装，例如，`kubectl scale`命令不过是专用于修改资源对象的replicas属性值而已，它也同样直接作用于活动对象。

不再有价值的活动对象可使用“`kubectl delete`”命令予以删除，需要删除Service对象myapp时，使用如下命令即可完成：

```
~]$ kubectl delete service myapp
service "myapp" deleted
```

有时候需要清空某一类型下的所有对象，只需要将上面命令对象的名称换成“`--all`”选项便能实现。例如，删除默认名称空间中所有的Deployment控制器的命令如下：

```
~]$ kubectl delete deployment --all
deployment.extensions "myapp" deleted
```

需要注意的是，受控于控制器的Pod对象在删除后会被重建，删除此类对象需要直接删除其控制器对象。不过，删除控制器时若不想删除其Pod对象，可在删除命令上使用“`--cascade=false`”选项。

虽然直接命令式管理的相关功能强大且适合用于操纵Kubernetes资源对象，但其明显的缺点是缺乏操作行为以及待运行对象的可信

源。另外，直接命令式管理资源对象存在较大的局限性，它们在设置资源对象属性方面提供的配置能力相当有限，而且还有不少资源并不支持命令操作进行创建，例如，用户无法创建带有多个容器的**Pod**对象，也无法为**Pod**对象创建存储卷。因此，管理资源对象更有效的方式是基于保存有对象配置信息的配置清单来进行。

2.5 本章小结

本章着重介绍了Kubernetes的三个核心资源抽象Pod、Deployment和Service，并在介绍了kubectl的基本用法之后通过案例讲解了如何在集群中部署、暴露、访问及扩缩容容器化应用，具体如下。

- Pod是运行容器化应用及调度的原子单元，同一个Pod中可同时运行多个容器，这些容器共享Mount、UTS及Network等Linux内核名称空间，并且能够访问同一组存储卷。

- Deployment是最常见的无状态应用的控制器，它支持应用的扩缩容、滚动更新等操作，为容器化应用赋予了极具弹性的功能。

- Service为弹性变动且存在生命周期的Pod对象提供了一个固定的访问接口，用于服务发现和服务访问。

- kubectl是Kubernetes API Server最常用的客户端程序之一，它功能强大，特性丰富，几乎能够完成除了安装部署之外的所有管理操作。

第3章 资源管理基础

Kubernetes系统的API Server基于HTTP/HTTPS接收并响应客户端的操作请求，它提供了一种“基于资源”（resource-based）的RESTful风格的编程接口，将集群的各种组件都抽象成为标准的REST资源，如Node、Namespace和Pod等，并支持通过标准的HTTP方法以JSON为数据序列化方案进行资源管理操作。本章将着重描述Kubernetes的资源管理方式。

3.1 资源对象及API群组

REST是Representational State Transfer的缩写，意为“表征状态转移”，它是一种程序架构风格，基本元素为资源（resource）、表征（representation）和行为（action）。资源即对象，一个资源通常意味着一个附带类型和关联数据、支持的操作方法以及与其他对象的关系的对象，它们是持有状态的事物，即REST中的S（State）。REST组件通过使用“表征”来捕获资源的当前或预期状态并在组件之间传输该表征从而对资源执行操作。表征是一个字节序列，由数据、描述数据的元数据以及偶尔描述元数据的元数据组成（通常用于验证消息的完整性），表征还有一些其他常用但不太精确的名称，如文档、文件和HTTP消息实体等。表征的数据格式称为媒体类型（media type），常用的有JSON或XML。API客户端不能直接访问资源，它们需要执行“动作”（action）来改变资源的状态，于是资源的状态从一种形式“转移”（Transfer）为另一种形式。

资源可以分组为集合（collection），每个集合只包含单一类型的资源，并且各资源间是无序的。当然，资源也可以不属于任何集合，它们称为单体资源。事实上，集合本身也是资源，它可以部署于全局级别，位于API的顶层，也可以包含于某个资源中，表现为“子集合”。集合、资源、子集合及子资源间的关系如图3-1所示。

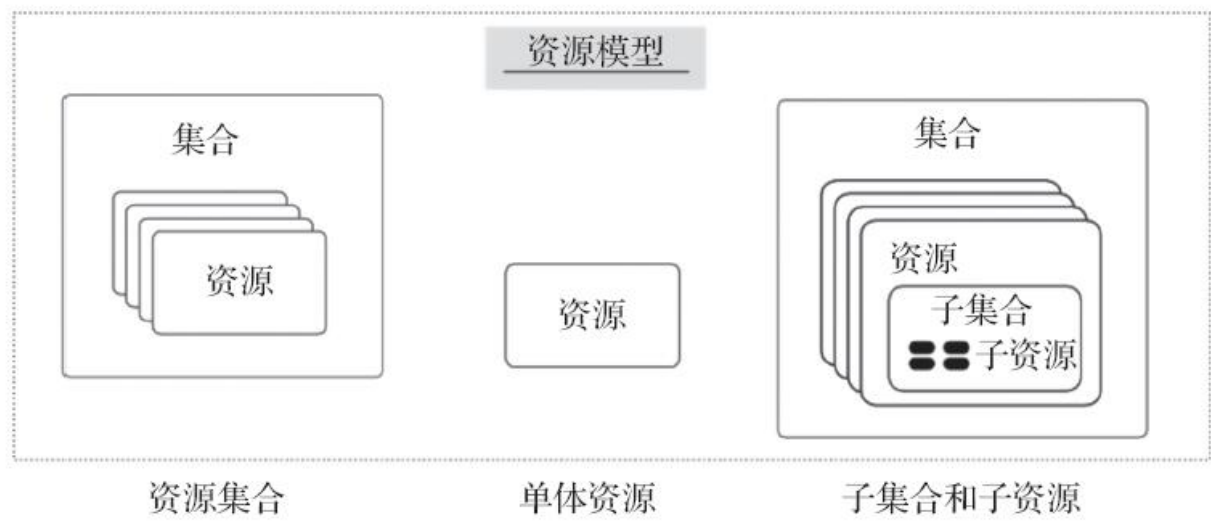


图3-1 集合、资源和子资源

Kubernetes系统将一切事物都抽象为API资源，其遵循REST架构风格组织并管理这些资源及其对象，同时还支持通过标准的HTTP方法（POST、PUT、PATCH、DELETE和GET）对资源进行增、删、改、查等管理操作。不过，在Kubernetes系统的语境中，“资源”用于表示“对象”的集合，例如，Pod资源可用于描述所有Pod类型的对象，但本书将不加区别地使用资源、对象和资源对象，并将它们统统理解为资源类型生成的实例—对象。

3.1.1 Kubernetes的资源对象

依据资源的主要功能作为分类标准，Kubernetes的API对象大体可分为工作负载（Workload）、发现和负载均衡（Discovery&LB）、配置和存储（Config&Storage）、集群（Cluster）以及元数据（Metadata）五个类别。它们基本上都是围绕一个核心目的而设计：

如何更好地运行和丰富Pod资源，从而为容器化应用提供更灵活、更完善的操作与管理组件，如图3-2所示。

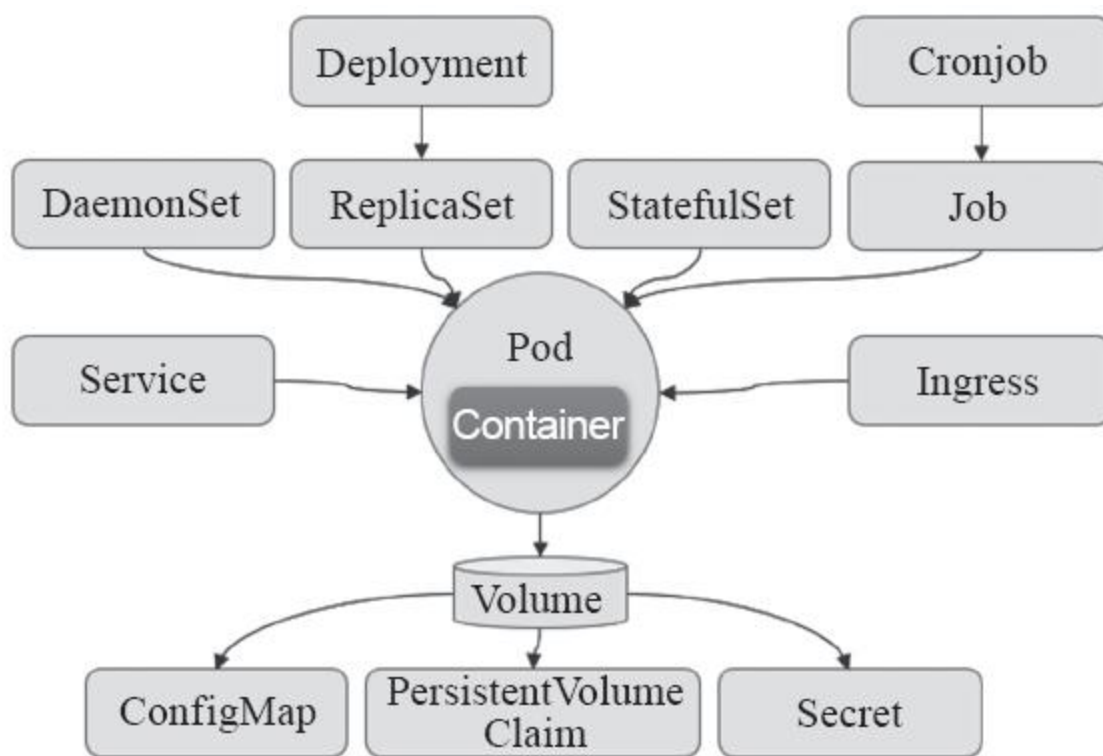


图3-2 Kubernetes常用资源对象

工作负载型资源用于确保Pod资源对象能够更好地运行容器化应用，具有同一种负载的各Pod对象需要以负载均衡的方式服务于各请求，而各种容器化应用彼此之间需要彼此“发现”以完成工作协同。Pod资源具有生命周期，存储型资源能够为重构的Pod对象提供持久化的数据存储机制，共享同一配置的Pod资源可从配置型资源中统一获取配置改动信息，这些资源作为配置中心为管理容器化应用的配置文件提供了极为便捷的管理机制。集群型资源为管理集群本身的工作特性

提供了配置接口，而元数据类型资源则用于配置集群内部的其他资源的行为。

(1) 工作负载型资源

Pod是工作负载型资源中的基础资源，它负责运行容器，并为其解决环境性的依赖，例如，向容器注入共享的或持久化的存储卷、配置信息或密钥数据等。但**Pod**可能会因为资源超限或节点故障等原因而终止，这些非正常终止的**Pod**资源需要被重建，不过，这类工作将由工作负载型的控制器来完成，它们通常也称为**pod**控制器。

应用程序分为**无状态**和**有状态**两种类型，它们对环境的依赖及工作特性有很大的不同，因此分属两种不同类型的**Pod**控制器来管理，**ReplicationController**、**ReplicaSet**和**Deployment**负责管理无状态应用，**StatefulSet**用于管控有状态类应用。**ReplicationController**是上一代的控制器，其功能由**ReplicaSet**和**Deployment**负责实现，因此几近于废弃。还有些应用较为独特，它们需要在集群中的每个节点上运行单个**Pod**资源，负责收集日志或运行系统服务等任务，这些**Pod**资源的管理则属于**DaemonSet**控制器的分内之事。另外，有些容器化应用需要继续运行以为守护进程不间断地提供服务，而有些则应该在正常完成后退出，这些在正常完成后就应该退出的容器化应用则由**Job**控制器负责管控。下面是各**Pod**控制器更为详细的说明。

- ReplicationController**: 用于确保每个**Pod**副本在任一时刻均能满足目标数量，换言之，它用于保证每个容器或容器组总是运行并且可访问；它是上一代的无状态**Pod**应用控制器，建议读者使用新型控制器**Deployment**和**ReplicaSet**来取代它。

- ReplicaSet**: 新一代**ReplicationController**，它与**ReplicationController**的唯一不同之处仅在于支持的标签选择器不同，**ReplicationController**只支持等值选择器，而**ReplicaSet**还额外支持基于集合的选择器。

- Deployment**: 用于管理无状态的持久化应用，例如**HTTP**服务器；它用于为**Pod**和**ReplicaSet**提供声明式更新，是建构在**ReplicaSet**之上的更为高级的控制器。

·**StatefulSet**: 用于管理有状态的持久化应用，如database服务程序；其与**Deployment**的不同之处在于**StatefulSet**会为每个Pod创建一个独有的持久性标识符，并确保各Pod之间的顺序性。

·**DaemonSet**: 用于确保每个节点都运行了某Pod的一个副本，新增的节点一样会被添加此类Pod；在节点移除时，此类Pod会被回收；**DaemonSet**常用于运行集群存储守护进程—如glusterd和ceph，还有日志收集进程—如fluentd和logstash，以及监控进程—如Prometheus的Node Exporter、collectd、Datadog agent和Ganglia的gmond等。

·**Job**: 用于管理运行完成后即可终止的应用，例如批处理作业任务；换句话讲，**Job**创建一个或多个Pod，并确保其符合目标数量，直到Pod正常结束而终止。

(2) 发现和负载均衡

Pod资源可能会因为任何意外故障而被重建，于是它需要固定的可被“发现”的方式。另外，**Pod**资源仅在集群内可见，它的客户端也可能是集群内的其他**Pod**资源，若要开放给外部网络中的用户访问，则需要事先将其暴露到集群外部，并且要为同一种工作负载的访问流量进行负载均衡。**Kubernetes**使用标准的资源对象来解决此类问题，它们是为工作负载添加发现机制及负载均衡功能的**Service**资源和**Endpoint**资源，以及通过七层代理实现请求流量负载均衡的**Ingress**资源。

(3) 配置与存储

Docker容器分层联合挂载的方式决定了不宜在容器内部存储需要持久化的数据，于是它通过引入挂载外部存储卷的方式来解决此类问题，而**Kubernetes**则为此设计了**Volume**资源，它支持众多类型的存储设备或存储系统，如GlusterFS、CEPH RBD和Flocker等。另外，新版本的**Kubernetes**还支持通过标准的CSI（Container Storage Interface）统一存储接口以及扩展支持更多类型的存储系统。

另外，基于镜像构建容器应用时，其配置信息于镜像制作时焙入，从而为不同的环境定制配置就变得较为困难。**Docker**使用环境变量等作为解决方案，但这么一来就得于容器启动时将值传入，且无法在运行时修改。**ConfigMap**资源能够以环境变量或存储卷的方式接入

到Pod资源的容器中，并且可被多个同类的Pod共享引用，从而实现“一次修改，多处生效”。不过，这种方式不适于存储敏感数据，如私钥、密码等，那是另一个资源类型Secret的功能。

(4) 集群级资源

Pod、Deployment、Service和ConfigMap等资源属于名称空间级别，可由相应的项目管理员所管理。然而，Kubernetes还存在一些集群级别的资源，用于定义集群自身配置信息的对象，它们仅应该由集群管理员进行操作。集群级资源主要包含以下几种类型。

- Namespace**: 资源对象名称的作用范围，绝大多数对象都隶属于某个名称空间，默认时隶属于“default”。

- Node**: Kubernetes集群的工作节点，其标识符在当前集群中必须是唯一的。

- Role**: 名称空间级别的由规则组成的权限集合，可被RoleBinding引用。

- ClusterRole**: Cluster级别的由规则组成的权限集合，可被RoleBinding和ClusterRoleBinding引用。

- RoleBinding**: 将Role中的许可权限绑定在一个或一组用户之上，它隶属于且仅能作用于一个名称空间；绑定时，可以引用同一名称空间中的Role，也可以引用全局名称空间中的ClusterRole。

- ClusterRoleBinding**: 将ClusterRole中定义的许可权限绑定在一个或一组用户之上；它能够引用全局名称空间中的ClusterRole，并能通过Subject添加相关信息。

(5) 元数据型资源

此类资源对象用于为集群内部的其他资源配置其行为或特性，如HorizontalPodAutoscaler资源可用于自动伸缩工作负载类型的资源对象的规模，Pod模板资源可用于为pod资源的创建预制模板，而LimitRange则可为名称空间的资源设置其CPU和内存等系统级资源的数量限制等。



提示 一个应用通常需要多个资源的支撑，例如，使用Deployment资源管理应用实例（Pod）、使用ConfigMap资源保存应用配置、使用Service或Ingress资源暴露服务、使用Volume资源提供外部存储等。

本书后面篇幅的主体部分就展开介绍这些资源类型，它们是将容器化应用托管运行于Kubernetes集群的重要工具组件。

3.1.2 资源及其在API中的组织形式

在Kubernetes上，资源对象代表了系统上的持久类实体，Kubernetes用这些持久类实体来表达集群的状态，包括容器化的应用程序正运行于哪些节点，每个应用程序有哪些资源可用，以及每个应用程序各自的行为策略，如重启、升级及容错策略等。一个对象可能会包含多个资源，用户可对这些资源执行增、删、改、查等管理操作。Kubernetes通常利用标准的RESTful术语来描述API概念。

- 资源类型（resource type）是指在URL中使用的名称，如Pod、Namespace和服务等，其URL格式为“GROUP/VERSION/RESOURCE”，如apps/v1/deployment。

- 所有资源类型都有一个对应的JSON表示格式，称为“种类”（kind）；客户端创建对象必须以JSON提交对象的配置信息。

- 隶属于同一种资源类型的对象组成的列表称为“集合”（collection），如PodList。

- 某种类型的单个实例称为“资源”（resource）或“对象”（object），如名为pod-demo的Pod对象。

kind代表着资源对象所属的类型，如Namespace、Deployment、Service及Pod等，而这些资源类型大体又可以分为三个类别，具体如下。

- 对象（Object）类：对象表示Kubernetes系统上的持久化实体，一个对象可能包含多个资源，客户端可用它执行多种操作。Namespace、Deployment、Service及Pod等都属于这个类别。

- 列表（List）类：列表通常是指同一类型资源的集合，如PodLists、NodeLists等。

- 简单（Simple）类：常用于在对象上执行某种特殊操作，或者管理非持久化的实体，如/binding或/status等。

Kubernetes绝大多数的API资源类型都是“对象”，它们代表着集群中某个概念的实例。有一小部分的API资源类型为“虚拟”（virtual）类型，它们用于表达一类“操作”（operation）。所有的对象型资源都有一个独有的名称标识以实现其幂等的创建及获取操作，不过，虚拟型资源无须获取或不依赖于幂等性时也可以不使用专用标识符。

有些资源类型隶属于集群范畴，如Namespace和PersistentVolume，而多数资源类型则受限于名称空间，如Pod、Deployment和Service等。名称空间级别的资源的URL路径中含有其所属空间的名称，这些资源对象在名称空间被删除时会被一并删除，并且这些资源对象的访问也将受控于其所属的名称空间级别的授权审查。

Kubernetes将API分割为多个逻辑组合，称为API群组，它们支持单独启用或禁用，并能够再次分解。API Server支持在不同的群组中使用不同的版本，允许各组以不同的速度演进，而且也支持同一群组同时存在不同的版本，如apps/v1、apps/v1beta2和apps/v1beta1，也因此能够在不同的群组中使用同名的资源类型，从而能在稳定版本的群组及新的实验群组中以不同的特性同时使用同一个资源类型。群组化管理的API使得其可以更轻松地进行扩展。当前系统的API Server上的相关信息可通过“`kubectl api-versions`”命令获取。命令结果中显示的不少API群组在后续的章节中配置资源清单时会多次用到：

```
[root@master ~]# kubectl api-versions
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
apps/v1
apps/v1beta1
apps/v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling/v1
autoscaling/v2beta1
batch/v1
batch/v1beta1
certificates.k8s.io/v1beta1
events.k8s.io/v1beta1
extensions/v1beta1
networking.k8s.io/v1
policy/v1beta1
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
```

```
scheduling.k8s.io/v1beta1
storage.k8s.io/v1
storage.k8s.io/v1beta1
v1
```

Kubernetes的API以层级结构组织在一起，每个API群组表现为一个以“/apis”为根路径的REST路径，不过核心群组core有一个专用的简化路径“/api/v1”。目前，常用的API群组可归为如下两类。

- 核心群组（core group）：REST路径为/api/v1，在资源的配置信息apiVersion字段中引用时可以不指定路径，而仅给出版本，如“apiVersion: v1”。

- 命名的群组（named group）：REST路径为/apis/\$GROUP_NAME/\$VERSION，例如/apis/apps/v1，它在apiVersion字段中引用的格式为“apiVersion: \$GROUP_NAME/\$VERSION”，如“apiVersion: apps/v1”。

总结起来，名称空间级别的每一个资源类型在API中的URL路径表示都可简单抽象为形如“/apis/<group>/<version>/namespaces/<namespace>/<kind-plural>”的路径，如default名称空间中Deployment类型的路径为/apis/apps/v1/namespaces/default/deployments，通过此路径可获取到default名称空间中所有Deployment对象的列表：

```
~]$ kubectl get --raw /apis/apps/v1/namespaces/default/deployments | jq .
{
  "kind": "DeploymentList",
  "apiVersion": "apps/v1",
  .....,
  "items": [.....]
}
```

另外，Kubernetes还支持用户自定义资源类型，目前常用的方式有三种：一是修改Kubernetes源代码自定义类型；二是创建一个自定义的API Server，并将其聚合至集群中；三是使用自定义资源（Custom Resource Definition，CRD）。

3.1.3 访问Kubernetes REST API

以编程的方式访问Kubernetes REST API有助于了解其流式化的集群管理机制，一种常用的方式是使用curl作为HTTP客户端直接通过API Server在集群上操作资源对象模拟请求和响应的过程。不过，由kubeadm部署的Kubernetes集群默认仅支持HTTPS的访问接口，它需进行一系列的认证检查，好在用户也可以借助kubectl proxy命令在本地主机上为API Server启动一个代理网关，由它支持使用HTTP进行通信，其工作逻辑如图3-3所示。

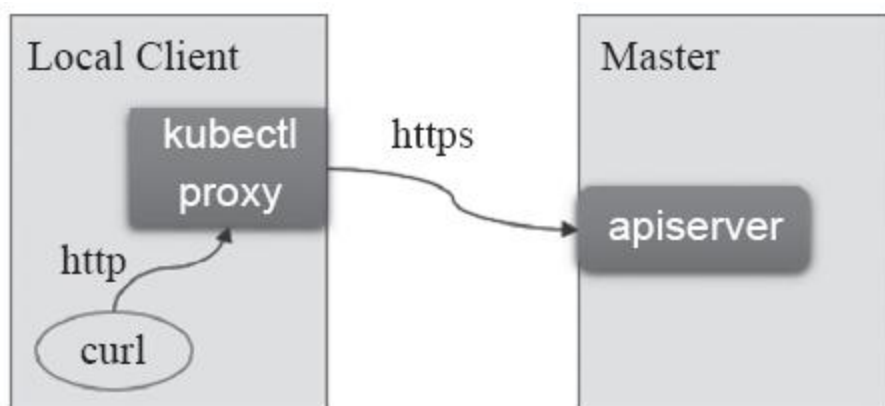


图3-3 kubectl在本地代理API Server

例如，于本地127.0.0.1的8080端口上启动API Server的一个代理网关：

```
~]$ kubectl proxy --port=8080
Starting to serve on 127.0.0.1:8080
```

而后即可于另一终端使用curl一类的客户端工具对此套接字地址发起访问请求，例如，请求Kubernetes集群上的NamespaceList资源对象，即列出集群上所有的Namespace对象：

```
~]$ curl localhost:8080/api/v1/namespaces/
{
  "kind": "NamespaceList",
  "apiVersion": "v1",
```

```
} .....
```

或者使用JSON的命令行处理器jq命令对响应的JSON数据流进行内容过滤，例如，下面的命令仅用于显示相关的NamespaceList对象中的各成员对象：

```
~]$ curl -s localhost:8080/api/v1/namespaces/ | jq .items[].metadata.name
"default"
"kube-public"
"kube-system"
```

给出特定的Namespace资源对象的名称则能够直接获取相应的资源信息，以kube-system名称空间为例：

```
~]$ curl -s localhost:8080/api/v1/namespaces/kube-system
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-system",
    "selfLink": "/api/v1/namespaces/kube-system",
    "uid": "eb6bf659-9d0e-11e8-bf0d-000c29ab0f5b",
    "resourceVersion": "33",
    "creationTimestamp": "2018-08-11T02:33:23Z"
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Active"
  }
}
```

上述命令响应的结果中展现了Kubernetes大多数资源对象的资源配置格式，它是一个JSON序列化的数据结构，具有kind、apiVersion、metadata、spec和status五个一级字段，各字段的意义和功能将在3.2节中重点介绍。

3.2 对象类资源格式

Kubernetes API仅接受及响应JSON格式的数据（JSON对象），同时，为了便于使用，它也允许用户提供YAML格式的POST对象，但API Server需要事先自行将其转换为JSON格式后方能提交。API Server接受和返回的所有JSON对象都遵循同一个模式，它们都具有kind和apiVersion字段，用于标识对象所属的资源类型、API群组及相关的版本。

进一步地，大多数的对象或列表类型的资源还需要具有三个嵌套型的字段metadata、spec和status。其中metadata字段为资源提供元数据信息，如名称、隶属的名称空间和标签等；spec则用于定义用户期望的状态，不同的资源类型，其状态的意义也各有不同，例如Pod资源最为核心的功能在于运行容器；而status则记录着活动对象的当前状态信息，它由Kubernetes系统自行维护，对用户来说为只读字段。

每个资源通常仅接受并返回单一类型的数据，而一种类型可以被多个反映特定用例的资源所接受或返回。例如对于Pod类型的资源来说，用户可创建、更新或删除Pod对象，然而，每个Pod对象的metadata、spec和status字段的值却又是各自独立的对象型数据，它们可被单独操作，尤其是status对象，是由Kubernetes系统单独进行自动更新，而不能由用户手动操作它。

3.2.1 资源配置清单

3.1节中曾使用curl命令通过代理的方式于API Server上请求到了kube-system这个Namespace活动对象的状态信息，事实上，用户也可以直接使用“`kubectl get TYPE/NAME-o yaml`”命令获取任何一个对象的YAML格式的配置文件，或者使用“`kubectl get TYPE/NAME-o json`”命令获取JSON格式的配置文件。例如，可使用下面的命令获取kube-system的状态：

```
~]$ kubectl get namespace kube-system -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: 2018-08-11T02:33:23Z
  name: kube-system
  resourceVersion: "33"
  selfLink: /api/v1/namespaces/kube-system
  uid: eb6bf659-9d0e-11e8-bf0d-000c29ab0f5b
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

除了极少数的资源之外，Kubernetes系统上的绝大多数资源都是由其使用者所创建的。创建时，需要以与上述输出结果中类似的方式以YAML或JSON序列化方案定义资源的相关配置数据，即用户期望的目标状态，而后再由Kubernetes的底层组件确保活动对象的运行时状态与用户提供的配置清单中定义的状态无限接近。因此，资源的创建要通过用户提供的资源配置清单来进行，其格式类似于kubectl get命令获取到的YAML或JSON形式的输出结果。不过，status字段对用户来说为只读字段，它由Kubernetes集群自动维护。例如，下面就是一个创建Namespace资源时提供的资源配置清单示例，它仅提供了几个必要的字段：

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
spec:
  finalizers:
  - kubernetes
```

将如上所述配置清单中的内容保存于文件中，使用“**kubectl create-f/PATH/TO/FILE**”命令即可将其创建到集群中。创建完成后查看其YAML或JSON格式的输出结果，可以看到Kubernetes会补全其大部分的字段，并提供相应的数据。事实上，Kubernetes的大多数资源都能够以类似的方式进行创建和查看，而且它们几乎都遵循类似的组织结构，下面的命令显示了第2章中使用**kubectl run**命令创建的Deployment资源对象**myapp**的状态信息：

```
~]$ kubectl get deployment myapp -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
.....
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      run: myapp
.....
status:
.....
```

为了节约篇幅，上面的输出结果省去了大部分内容，仅保留了其主体结构。从命令结果可以看出，它也遵循Kubernetes API标准的资源组织格式，由**apiVersion**、**kind**、**metadata**、**spec**和**status**五个核心字段组成，只是**spec**字段中嵌套的内容与Namespace资源几乎完全不同。

事实上，对几乎所有的资源来说，**apiVersion**、**kind**和**metadata**字段的功能基本上都是相同的，但**spec**则用于资源的期望状态，而资源之所以存在类型上的不同，也在于它们的嵌套属性存在显著差别，它由用户定义和维护。而**status**字段则用于记录活动对象的当前状态，它要与用户在**spec**中定义的期望状态相同，或者正处于转换为与其相同的过程中。

3.2.2 metadata嵌套字段

metadata字段用于描述对象的属性信息，其内嵌多个字段用于定义资源的元数据，例如**name**和**labels**等，这些字段大体可分为必要字段和可选字段两大类。名称空间级别的资源的必选字段包括如下三项。

- namespace**: 指定当前对象隶属的名称空间，默认值为**default**。

- name**: 设定当前对象的名称，在其所属的名称空间的同一类型中必须唯一。

- uid**: 当前对象的唯一标识符，其唯一性仅发生在特定的时间段和名称空间中；此标识符主要是用于区别拥有同样名字的“已删除”和“重新创建”的同一个名称的对象。

可选字段通常是指由**Kubernetes**系统自行维护和设置，或者存在默认，或者本身允许使用空值等类型的字段，常用的有如下几个：

- labels**: 设定用于标识当前对象的标签，键值数据，常被用作挑选条件。

- annotations**: 非标识型键值数据，用来作为挑选条件，用于**labels**的补充。

- resourceVersion**: 当前对象的内部版本标识符，用于让客户端确定对象变动与否。

- generation**: 用于标识当前对象目标状态的代别。

- creationTimestamp**: 当前对象创建日期的时间戳。

- deletionTimestamp**: 当前对象删除日期的时间戳。

此外，用户通过配置清单创建资源时，通常仅需要给出必选字段，可选字段可按需指定，对于用户未明确定义的嵌套字段，则需要由一系列的**finalizer**组件自动予以填充。而用户需要对资源创建的目标资源对象进行强制校验，或者在修改时需要用到**initializer**组件完成，

例如，为每个待创建的Pod对象添加一个Sidecar容器等。不同的资源类型也会存在一些专有的嵌套字段，例如，ConfigMap资源还支持使用clusterName等。

3.2.3 spec和status字段

Kubernetes用spec来描述所期望的对象应该具有的状态，而用status字段来记录对象在系统上的当前状态，因此status字段仅对活动对象才有意义。这两个字段都属于嵌套类型的字段。在定义资源配置清单时，spec是必须定义的字段，用于描述对象的目标状态，即用户期望对象需要表现出来的特征。status字段则记录了对象的当前状态（或实际状态），此字段值由Kubernetes系统负责填充或更新，用户不能手动进行定义。Master的controller-manager通过相应的控制器组件动态管理并确保对象的实际状态匹配用户所期望的状态，它是一种调和（reconciliation）配置系统。

例如，Deployment是一种用于描述集群中运行的应用的对象，因此，创建Deployment类型的对象时，需要为目标Deployment对象设定spec，指定期望需要运行的Pod副本数量、使用的标签选择器以及Pod模板等。Kubernetes系统读取待创建的Deployment对象的spec以及系统上相应的活动对象的当前状态，必要时进行对象更新以确保status字段吻合spec字段中期望的状态。如果这其中任一实例出现问题（status字段值发生了变化），那么Kubernetes系统则需要及时对spec和status字段的差异做出响应，例如，补足缺失的Pod副本数目等。

spec字段嵌套的字段对于不同的对象类型来说各不相同，具体需要参照Kubernetes API参考手册中的说明分别进行获取，核心资源对象的常用配置字段将会在本书后面的章节中进行讲解。

3.2.4 资源配置清单格式文档

定义资源配置清单时，尽管apiVersion、kind和metadata有章可循，但spec字段对不同的资源来说却是千差万别的，因此用户需要参考Kubernetes API的参考文档来了解各种可用属性字段。好在，Kubernetes在系统上内建了相关的文档，用户可以使用“`kubectl explain`”命令直接获取相关的使用帮助，它将根据给出的对象类型或相应的嵌套字段来显示相关的下一级文档。例如，要了解Pod资源的一级字段，可以使用类似如下的命令，命令结果会输出支持使用的各一组字段及其说明：

```
~]$ kubectl explain pods
```

需要了解某一级字段表示的对象之下的二级对象字段时，只需要指定其一级字段的对象名称即可，三级和四级字段对象等的查看方式依此类推。例如查看Pod资源的Spec对象支持嵌套使用的二级字段，可使用类似如下的命令：

```
~]$ kubectl explain pods.spec
RESOURCE: spec <Object>

DESCRIPTION:
    Specification of the desired behavior of the pod. ....

    PodSpec is a description of a pod.

FIELDS:
    activeDeadlineSeconds    <integer>
        Optional duration in seconds the pod may be active on the node relative to
        StartTime before the system will actively try to mark it failed and kill
        associated containers. Value must be a positive integer.
    ....
    containers    <[]Object> -required-
        List of containers belonging to the pod. Containers cannot currently be
        added or removed. There must be at least one container in a Pod. Cannot
        be updated.
    ....
```

对象的spec字段的文档通常包含RESOURCE、DESCRIPTION和FIELDS几节，其中FIELDS节中给出了可嵌套使用的字段、数据类型及功能描述。例如，上面命令的结果显示在FIELDS中的containers字段

的数据类型是一个对象列表（`[]Object`），而且是一个必选字段。任何值为对象类型数据的字段都会嵌套一到多个下一级字段，例如，**Pod**对象中的每个容器也是对象类型数据，它同样包含嵌套字段，但容器不支持单独创建，而是要包含于**Pod**对象的上下文中，其详细信息可通过三级字段来获取，命令及其结果示例如下：

```
~]$ kubectl explain pods.spec.containers
RESOURCE: containers <[]Object>

DESCRIPTION:
    List of containers belonging to the pod. Containers cannot currently be
    added
    or removed. There must be at least one container in a Pod. Cannot be updated.

    A single application container that you want to run within a pod.

FIELDS:
    args <[]string>
        Arguments to the entrypoint. The docker image's CMD is used if this is not
        provided. ....

    command <[]string>
        Entrypoint array. Not executed within a shell. The docker image's
        ENTRYPOINT is used if this is not provided. ....

    env <[]Object>
        List of environment variables to set in the container. Cannot be updated.

    .....
```

内建文档大大降低了用户手动创建资源配置清单的难度，尝试使用某个资源类型时，**explain**也是用户的常用命令之一。熟悉各常用字段的功用之后，以同类型的现有活动对象的清单为模板可以更快地生成目标资源的配置文件，命令格式为“**kubectl get TYPE NAME -o yaml --export**”，其中**--export**选项用于省略输出由系统生成的信息。例如，基于现在的**Deployment**资源对象**myapp**生成配置模板**deploy-demo.yaml**文件，可以使用如下命令：

```
~]$ kubectl get deployment myapp -o yaml --export > deploy-demo.yaml
```

通过资源清单文件管理资源对象较之直接通过命令行进行操作有着诸多优势，具体包括命令行的操作方式仅支持部分资源对象的部分属性，而资源清单支持配置资源的所有属性字段，而且使用配置清单

文件还能够进行版本追踪、复审等高级功能的操作。本书后续章节中的大部分资源管理操作都会借助于资源配置文件进行。

3.2.5 资源对象管理方式

Kubernetes的API Server遵循声明式编程（**declarative programming**）范式而设计，侧重于构建程序逻辑而无须用户描述其实现流程，用户只需要设定期望的状态，系统即能自行确定需要执行的操作以确保达到用户期望的状态。例如，期望某Deployment控制器管理三个Pod资源对象时，而系统观察到的当前数量却是两个，于是系统就会知道需要创建一个新的Pod资源来满足此期望。Kubernetes的自愈、自治等功能都依赖于其声明式机制。

与此对应的另一种范式称为陈述式编程（**imperative programming**），代码侧重于通过创建一种告诉计算机如何执行操作的算法来更改程序状态的语句来完成，它与硬件的工作方式密切相关，通常，代码将使用条件语句、循环和类继承等控制结构。为了便于用户使用，Kubernetes的API Server也支持陈述式范式，它直接通过命令及其选项完成对象的管理操作，前面用到的run、expose、delete和get等命令都属于此类，执行时用户需要告诉系统要做什么，例如，使用run命令创建一个有着3个Pod对象副本的Deployment对象，或者通过delete命令删除一个名为myapp的Service对象。

Kubernetes系统的大部分API对象都有着spec和status两个字段，其中，spec用于让用户定义所期望的状态，系统从中读出相关的定义；而status则是系统观察并负责写入的当前状态，用户可以从获取相关的信息。Kubernetes系统通过控制器监控着系统对象，由其负责让系统当前的状态无限接近用户所期望的状态。

kubectl的命令由此可以分为三类：陈述式命令（**imperative command**）、陈述式对象配置（**imperative object configuration**）和声明式对象配置（**declarative object configuration**）。第一种方式即此前用到的run、expose、delete和get等命令，它们直接作用于Kubernetes系统上的活动对象，简单易用，但不支持代码复用、修改复审及审计日志等功能，这些功能的使用通常要依赖于资源配置文件，这些文件也称为资源清单。在这种模式下，用户可以访问每个对象的完整模式，但用户还需要深入学习Kubernetes API。

如3.2.4节所述，资源清单本质上是一个JSON或YAML格式的文本文件，由资源对象的配置信息组成，支持使用Git等进行版本控制。而用户可以资源清单为基础，在Kubernetes系统上以陈述式或声明式进行资源对象管理，如图3-4所示。

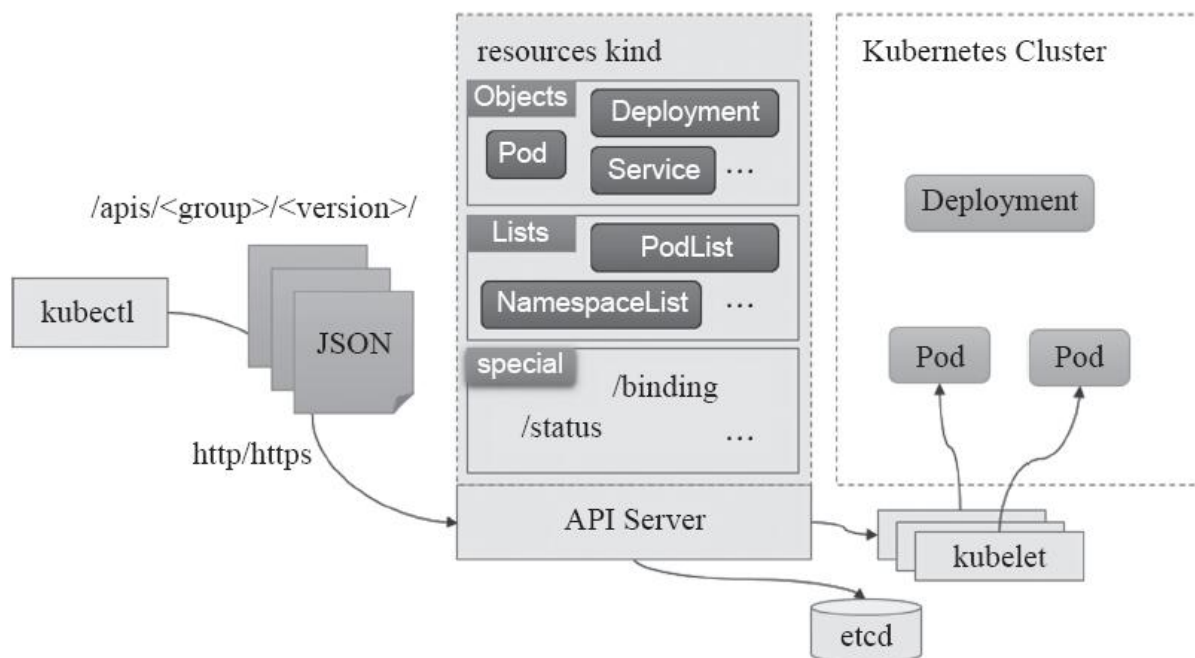


图3-4 基于资源配置清单管理对象

陈述式管理方式包括create、delete、get和replace等命令，与陈述式命令的不同之处在于，它通过资源配置清单读取需要管理的目标资源对象。陈述式对象配置的管理操作直接作用于活动对象，即便仅修改配置清单中的极小一部分内容，使用replace命令进行的对象更新也将会导致整个对象被替换。进一步地，混合使用陈述式命令进行清单文件带外修改时，必然会导致用户丢失活动对象的当前状态。

声明式对象配置并不直接指明要进行的对象管理操作，而是提供配置清单文件给Kubernetes系统，并委托系统跟踪活动对象的状态变动。资源对象的创建、删除及修改操作全部通过唯一的命令apply来完成，并且每次操作时，提供给命令的配置信息都将保存于对象的注解信息（`kubectl.kubernetes.io/last-applied-configuration`）中，并通过对比检查活动对象的当前状态、注解中的配置信息及资源清单中的配置信息三方进行变更合并，从而实现仅修改变动字段的高级补丁机制。

陈述式对象配置相较于声明式对象配置来说，其缺点在于同一目录下的配置文件必须同时进行同一种操作，例如，要么都创建，要么都更新等，而且其他用户的更新也必须反映在配置文件中，不然其更新在一下次的更新中将会被覆盖。因此，声明式对象配置是优先推荐给用户使用的管理机制。

然而，对于新手来说，陈述式命令的配置方式最易于上手，对系统有所了解后易于切换为使用陈述式对象配置管理方式。因此，若推荐给高级用户则推荐使用声明式配置，并建议同时使用版本控制系统存储所期望的状态，以及跨对象的历史信息，并启用变更复审机制。另外，推荐使用借助于**kube-applier**等一类的项目实现自动化声明式配置，用户将配置推送到**Git**仓库中，然后借助此类工具即能将其自动同步于**Kubernetes**集群上。

3.3 kubectl命令与资源管理

API Server是Kubernetes集群的网关，用户和管理员以及其他客户端仅能通过此网关接口与集群进行交互。API是面向程序员的访问接口，目前可较好地支持Golang和Python编程语言，当然，终端用户更为常用的是通用命令行工具kubectl。访问集群之前，各类客户端需要了解集群的位置并拥有访问集群的凭据才能获取访问许可。使用kubeadm进行集群初始化时，kubeadm init自动生成的/etc/kubernetes/admin.conf文件是客户端接入当前集群时使用的kubeconfig文件，它内建了用于访问Kubernetes集群的最高管理权限的用户账号及相关的认证凭据，可由kubectl直接使用。

3.3.1 资源管理操作概述

资源的管理操作可简单归结为增、删、改、查四种，`kubectl`提供了一系列的子命令用于执行此类任务，如`create`、`delete`、`patch`、`apply`、`replace`、`edit`、`get`等，其中有些命令必须基于资源清单来进行，如`apply`和`replace`命令，也有些命令既可基于清单文件进行，也可实时作用于活动资源之上，如`create`、`get`、`patch`和`delete`等。

`kubectl`命令能够读取任何以`.yaml`、`.yml`或`.json`为后缀的文件（可称为配置清单或配置文件，后文将不加区别地使用这两个术语）。实践中，用户既可以为每个资源使用专用的清单文件，也可以将多个相关的资源（例如，属于同一个应用或微服务）组织在同一个清单文件中。不过，如果是YAML格式的清单文件，多个资源彼此之间要使用“`---`”符号作为单独的一行进行资源分割。这样，多个资源就将以清单文件中定义的次序被`create`、`apply`等子命令调用。

`kubectl`的多数子命令支持使用“`-f`”选项指定使用的清单文件路径或URL，也可以是存储有清单文件的目录，另外，此选项在同一命令中也可重复使用多次。如果指定的目录路径存在子目录中时，那么可按需同时使用“`-R`”选项以递归获取子目录中的配置清单。

再者，支持使用标签和注解是Kubernetes系统的一大特色，它为资源管理机制增色不少，而且`delete`和`get`等命令能够基于标签挑选目标对象，有些资源甚至必须依赖于标签才能正常使用和工作，例如Service和Pod控制器Deployment等资源对象。子命令`label`用于管理资源标签，而管理资源注解的子命令则是`annotate`。

就地更新（修改）现有的资源也是一种常见的操作。`apply`命令通过比较资源在清单文件中的版本及前一次的版本执行更新操作，它不会对未定义的属性产生额外的作用。`edit`命令相当于先使用`get`命令获取资源配置，通过交互式编辑器修改后再自动使用`apply`命令将其应用。`patch`命令基于JSON补丁、JSON合并补丁及策略合并补丁对资源进行就地更新操作。



提示 为了利用`apply`命令的优势，用户应该总是使用`apply`命令或`create--save-config`命令创建资源。

3.3.2 kubectl的基本用法

kubectl是最常用的客户端工具之一，它提供了基于命令行访问Kubernetes API的简洁方式，能够满足对Kubernetes的绝大部分的操作需求。例如，需要创建资源对象时，kubectl会将JSON格式的清单内容以POST方式提交至API Server。本节主要描述kubectl的基本功能。



提示 如果要单独部署kubectl，Kubernetes也提供了相应的单独发行包，或者适配于各平台的程序管理器的相关程序包，如rpm包或deb包等，用户根据平台类型的不同获取相匹配的版本安装完成即可，操作步骤类似于前面的安装方法，因此这里不再给出其具体过程。

kubectl是Kubernetes系统的命令行客户端工具，特性丰富且功能强大，是Kubernetes管理员最常用的集群管理工具。其最基本的语法格式为“kubectl[command][TYPE][NAME][flags]”，其中各部分的简要说明如下。

1) **command**: 对资源执行相应操作的子命令，如get、create、delete、run等；常用的核心子命令如表3-1所示。

2) **TYPE**: 要操作的资源对象的类型，如pods、services等；类型名称区分字符大小写，但支持使用简写格式。

3) **NAME**: 对象名称，区分字符大小写；省略时，则表示指定TYPE的所有资源对象；另外，也可以使用“TYPE/NAME”的格式来表示资源对象。

4) **flags**: 命令行选项，如“-s”或“--server”；另外，get等命令在输出时还有一个常用的标志“-o<format>”用于指定输出格式，如表3-1所示。

表3-1 kubectl的子命令列表

命令	命令类别	功能说明
create	基础命令（初级）	通过文件或标准输入创建资源
expose		基于 rc、service、deployment 或 pod 创建 Service 资源
run		通过创建 Deployment 在集群中运行指定的镜像
set		设置指定资源的特定属性
get	基础命令（中级）	显示一个或多个资源
explain		打印资源文档
edit		编辑资源
delete		基于文件名、stdin、资源或名字，以及资源和选择器删除资源
rollout	部署命令	管理资源的滚动更新
rolling-update		对 ReplicationController 执行滚动升级
scale		伸缩 Deployment、ReplicaSet、RC 或 Job 的规模
autoscale		对 Deployment、ReplicaSet 或 RC 进行自动伸缩
certificate	集群管理命令	配置数字证书资源
cluster-info		打印集群信息
top		打印资源（CPU/Memory/Storage）使用率
cordon		将指定 node 设定为“不可用”(unschedulable) 状态
uncordon		将指定 node 设定为“可用”(schedulable) 状态
drain		“排干”指定的 node 的负载以进入“维护”模式
taint		为 node 声明污点及标准行为
describe	排错及调试命令	显示指定的资源或资源组的详细信息
logs		显示一个 Pod 内某容器的日志
attach		附加终端至一个运行中的容器
exec		在容器中执行指定命令
port-forward		将本地的一个或多个端口转发至指定的 Pod
proxy		创建能够访问 Kubernetes API Server 的代理
cp		在容器间复制文件或目录
auth		打印授权信息

(续)

命令	命令类别	功能说明
apply	高级命令	基于文件或 stdin 将配置应用于资源
patch		使用策略合并补丁更新资源字段
replace		基于文件或 stdin 替换一个资源
convert		为不同的 API 版本转换配置文件
label	设置命令	更新指定资源的 label
annotate		更新资源的 annotation
completion		输出指定的 shell (如 bash) 的补全码
version	其他命令	打印 Kubernetes 服务端和客户端的版本信息
api-versions		以 “group/version” 格式打印服务器支持的 API 版本信息
config		配置 kubeconfig 文件的内容
plugin		运行命令行插件
help		打印任一命令的帮助信息

kubectl命令还包含了多种不同的输出格式（如表3-2所示），它们为用户提供了非常灵活的自定义输出机制，如输出为YAML或JSON格式等。

表3-2 kubectl get命令的常用输出格式

输出格式	格式说明
-o wide	显示资源的额外信息
-o name	仅打印资源的名称
-o yaml	YAML 格式化输出 API 对象信息
-o json	JSON 格式化输出 API 对象信息
-o go-template	以自定义的 go 模板格式化输出 API 对象信息
-o custom-columns	自定义要输出的字段

此外，kubectl命令还有许多通用的选项，这个可以使用“kubectl options”命令来获取。下面列举几个比较常用命令。

- s或--server: 指定API Server的地址和端口。
- kubeconfig: 使用的kubeconfig文件路径，默认为~/.kube/config。
- namespace: 命令执行的目标名称空间。

kubectl的部分子命令在第2章已经多次提到，余下的大多数命令在后续的章节中还会用到，对于它们的使用说明也将在首次用到时进行

展开说明。

3.4 管理名称空间资源

名称空间（**Namespace**）是**Kubernetes**集群级别的资源，用于将集群分隔为多个隔离的逻辑分区以配置给不同的用户、租户、环境或项目使用，例如，可以为**development**、**qa**和**production**应用环境分别创建各自的名称空间。

Kubernetes的绝大多数资源都隶属于名称空间级别（另一个是全局级别或集群级别），名称空间资源为这类的资源名称提供了隔离的作用域，同一名称空间内的同一类型资源名必须是唯一的，但跨名称空间时并无此限制。不过，**Kubernetes**还是有一些资源隶属于集群级别的，如**Node**、**Namespace**和**PersistentVolume**等资源，它们不属于任何名称空间，因此资源对象的名称必须全局唯一。



注意 **Kubernetes**的名称空间资源不同于**Linux**系统的名称空间，它们是各自独立的概念。另外，**Kubernetes**名称空间并不能实现**Pod**间的通信隔离，它仅用于限制资源对象名称的作用域。

3.4.1 查看名称空间及其资源对象

Kubernetes集群默认提供了几个名称空间用于特定的目的，例如，**kube-system**主要用于运行系统级资源，而**default**则为那些未指定名称空间的资源操作提供一个默认值，前面章节中的绝大多数资源管理操作都在**default**名称空间中进行。“**kubectl get namespaces**”命令则可以查看**namespaces**资源：

```
~]$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    6d
kube-public   Active    6d
kube-system   Active    6d
```

也可以使用“**kubectl describe namespaces**”命令查看特定名称空间的详细信息，例如：

```
~]$ kubectl describe namespaces default
```

kubectl的资源查看命令在多数情况下应该针对特定的名称空间来进行，为其使用“-n”或“--namespace”选项即可，例如，查看**kube-system**下的所有Pod资源：

```
~]$ kubectl get pods -n kube-system
NAME                                READY    STATUS    RESTARTS    AGE
etcd-master.ikubernetes.io         1/1     Running   1           6d
kube-apiserver-master.ikubernetes.io 1/1     Running   1           6d
.....
```

命令结果显示**kube-system**与**default**名称空间的Pod资源对象并不相同，这正是**Namespace**资源的名称隔离功能的体现。有了**Namespace**对象，用户再也不必精心安排资源名称，也不用担心误操作了其他用户的资源。

3.4.2 管理Namespace资源

Namespace是Kubernetes API的标准资源类型之一，如3.2.1节中所述，它的配置主要有kind、apiVersion、metadata和spec等一级字段组成。将3.2.1节中的名称空间配置清单保存于配置文件中，使用陈述式对象配置命令create或声明式对象配置命令apply便能完成创建：

```
~]$ kubectl apply -f namespace-example.yaml
namespace/dev created
```

名称空间资源属性较少（通常只需要指定名称即可），简单起见，kubectl为其提供了一个封装的专用陈述式命令“**kubectl create namespace**”。Namespace资源的名称仅能由字母、数字、连接线、下划线等字符组成。例如，下面的命令可用于创建名为qa的Namespace对象：

```
~]$ kubectl create namespace qa
namespace/qa created
```



注意 namespace资源的名称仅能由字母、数字、连接线、下划线等字符组成。

实践中，不建议混用不同类别的管理方式。考虑到声明式对象配置管理机制的强大功能，强烈推荐用户使用apply和patch等命令进行资源创建及修改一类的管理操作。

使用kubectl管理资源时，如果一并提供了名称空间选项，就表示此管理操作仅针对指定名称空间进行，而删除Namespace资源会级联删除其包含的所有其他资源对象。表3-3给出了几个常用的命令格式。

表3-3 结合名称空间使用的删除命令

命令格式	功能
kubectl delete TYPE RESOURCE -n NS	删除指定名称空间内的指定资源
kubectl delete TYPE --all -n NS	删除指定名称空间内的指定类型的所有资源
kubectl delete all -n NS	删除指定名称空间内的所有资源
kubectl delete all --all	删除所有名称空间中的所有资源

需要再次指出的是，**Namespace**对象仅用于资源对象名称的隔离，它自身并不能隔绝跨名称空间的Pod间通信，那是网络策略（**network policy**）资源的功能。

3.5 Pod资源的基础管理操作

Pod是Kubernetes API中的核心资源类型，它可以定义在JSON或YAML格式的资源清单中，由资源管理命令进行陈述式或声明式管理。创建时，用户通过`create`或`apply`命令将请求提交到API Server并将其保存至集群状态存储系统etcd中，而后由调度器将其调度至最佳目标节点，并被相应节点的kubelet借助于容器引擎创建并启动。这种由用户直接通过API创建的Pod对象也称为自主式Pod。

3.5.1 陈述式对象配置管理方式

陈述式对象配置管理机制，是由用户通过配置文件指定要管理的目标资源对象，而后再由用户借助于命令直接指定Kubernetes系统要执行的管理操作的管理方式，常用的命令有create、delete、replace、get和describe等。

1. 创建Pod资源

Pod是标准的Kubernetes API资源，在配置清单中使用kind、apiVersion、metadata和spec字段进行定义，status字段在对象创建后由系统自行维护。Pod对象的核心功用在于运行容器化应用，在其spec字段中嵌套的必选字段是containers，它的值是一个容器对象列表，支持嵌套创建一到多个容器。下面是一个Pod资源清单示例文件，在spec中定义的期望的状态是在Pod对象中基于ikubernetes/myapp: v1镜像运行一个名为myapp的容器：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
```

把上面的内容保存于配置文件中，使用“kubectl[COMMAND]-f/PATH/TO/YAML_FILE”命令以陈述式对象配置进行资源对象的创建，下面是相应的命令及响应结果：

```
~]$ kubectl create -f pod-example.yaml
pod/pod-example created
```



提示 如果读者熟悉JSON，也可以直接将清单文件定义为JSON格式；YAML格式的清单文件本身也是由API Server事先将其转换为JSON格式而后才进行应用的。

命令的返回信息表示目标Pod对象pod-example得以成功创建。事实上，**create**命令中的**-f**选项也支持使用目录路径或URL，而且目标路径为目录时，还支持使用**-R**选项进行子目录递归。另外，**--record**选项可以将命令本身记录为目标对象的注解信息kubernetes.io/change-cause，而**--save-config**则能够将提供给命令的资源对象配置信息保存于对象的注解信息kubectl.kubernetes.io/last-applied-configuration中，后一个命令的功用与声明式对象配置命令**apply**的功能相近。

2.查看Pod状态

get命令默认显示资源对象最为关键的状态信息，而**describe**等命令则能够打印出Kubernetes资源对象的详细状态。不过，虽然创建时给出的资源清单文件较为简洁，但“**kubectl get**”命令既可以使用“**-o yaml**”或“**-o json**”选项输出资源对象的配置数据及状态，也能够借助于“**--custom-columns**”选项自定义要显示的字段：

```
~]$ kubectl get -f pod-example.yaml
NAME          READY    STATUS    RESTARTS   AGE
pod-example   1/1      Running   0           1m
~]$ kubectl get -f pod-example.yaml -o custom-
columns=NAME:metadata.name,STATUS:status.phase
NAME          STATUS
pod-example   Running
```

使用“**-o yaml**”或“**-o json**”选项时，**get**命令能够返回资源对象的元数据、期望的状态及当前状态数据信息，而要打印活动对象的详细信息，则需要**describe**命令，它可根据资源清单、资源名或卷标等方式过滤输出符合条件的资源对象的信息。命令格式为“**kubectl describe (-f FILENAME|TYPE[NAME_PREFIX|-l label]|TYPE/NAME)**”。例如，显示pod-example的详细信息，可使用类似如下的命令：

```
~]$ kubectl describe -f pod-example.yaml
```

对于Pod资源对象来说，它能够返回活动对象的元数据、当前状态、容器列表及各容器的详情、存储卷对象列表、QoS类别、事件及相关信息，这些详情对于了解目标资源对象的状态或进行错误排查等操作来说至关重要。

3.更新Pod资源

对于活动对象，并非其每个属性值都支持修改，例如，Pod资源对象的`metadata.name`字段就不支持修改，除非删除并重建它。对于那些支持修改的属性，比如，容器的`image`字段，可将其完整的配置清单导出于配置文件中并更新相应的配置数据，而后使用`replace`命令基于陈述式对象配置的管理机制进行资源对象的更新。例如，将前面创建`pod-example`时使用的资源清单中的`image`值修改为“`ikubernetes/myapp: v2`”，而后执行更新操作：

```
~]$ kubectl get pods pod-example -o yaml > pod-example-update.yaml
~]$ sed -i 's@\(\image:\)\.*@ikubernetes/myapp:v2@' pod-example-update.yaml
~]$ kubectl replace -f pod-example-update.yaml
pod/pod-example replaced
```

更新活动对象的配置时，`replace`命令要重构整个资源对象，故此它必须基于完整格式的配置信息才能进行活动对象的完全替换。若要基于此前的配置文件进行替换，就必须使用`--force`选项删除此前的活动对象，而后再进行新建操作，否则命令会返回错误信息。例如，将前面第一步“创建Pod资源”内的配置清单中的镜像修改为“`ikubernetes/myapp: v2`”后再进行强制替换，命令如下：

```
~]$ kubectl replace -f pod-example.yaml --force
pod "pod-example" deleted
pod/pod-example replaced
```

4.删除Pod资源

陈述式对象配置管理方式下的删除操作与创建、查看及更新操作类似，为`delete`命令使用`-f`选项指定配置清单即可，例如，删除`pod-example.yaml`文件中定义的Pod资源对象：

```
~]$ kubectl delete -f pod-example.yaml
pod "pod-example" deleted
```

之后再次打印相关配置清单中定义的资源对象即可验证其删除的结果，例如：

```
~]$ kubectl get -f pod-example.yaml
No resources found.
Error from server (NotFound): pods "pod-example" not found
```

3.5.2 声明式对象配置管理方式

陈述式对象配置管理机制中，同时指定的多个资源必须进行同一种操作，而且其**replace**命令是通过完全替换现有的活动对象来进行资源的更新操作，对于生产环境来说，这并非理想的选择。声明式对象配置操作在管理资源对象时将配置信息保存于目标对象的注解中，并通过比较活动对象的当前配置、前一次管理操作时保存于注解中的配置，以及当前命令提供的配置生成更新补丁从而完成活动对象的补丁式更新操作。此类管理操作的常用命令有**apply**和**patch**等。

例如，创建3.5.1节中定义的主容器使用“**ikubernetes/myapp: v1**”镜像的**Pod**资源对象，还可以使用如下命令进行：

```
~]$ kubectl apply -f pod-example.yaml
pod/pod-example created
```

而更新对象的操作，可在直接修改原有资源清单文件后再次对其执行**apply**命令来完成，例如，修改**Pod**资源配置清单中的镜像文件为“**ikubernetes/myapp: v2**”后再次执行如上的**apply**命令：

```
~]$ kubectl apply -f pod-example.yaml
pod/pod-example configured
```

命令结果显示资源重新配置完成并且已经生效。事实上，此类操作也完全能够使用**patch**命令直接进行补丁操作。而资源对象的删除操作依然可以使用**apply**命令，但要同时使用**--prune**选项，命令的格式为“**kubectl apply-f<directory/>--prune-l<labels>**”。需要注意的是，此命令异常凶险，因为它将基于标签选择器过滤出所有符合条件的对象，并检查由**-f**指定的目录中是否存在某配置文件已经定义了相应的资源对象，那些不存在相应定义的资源对象将被删除。因此，删除资源对象的操作依然建议使用陈述式对象配置方式的命令“**kubectl delete**”进行，这样的命令格式操作目标明确且不易出现偏差。

3.6 本章小结

本章介绍了Kubernetes系统上常用的资源对象类型及其管理方式，在说明kubectl命令行工具的基础用法后借助于Namespace资源对象简单说明了其使用方式，具体如下。

- Kubernetes提供了RESTful风格的API，它将各类组件均抽象为“资源”，并通过属性赋值完成实例化。

- Kubernetes API支持的资源类型众多，包括Node、Namespace、Pod、Service、Deployment、ConfigMap等上百种。

- 标准格式的资源配置大多都是由kind、apiVersion、metadata、spec和status等一级属性字段组成，其中spec是由用户定义的期望状态，而status则是由系统维护的当前状态。

- Kubernetes API主要提供的是声明式对象配置接口，但它也支持陈述式命令及陈述式对象配置的管理方式。

- kubectl命令功能众多，它将通过子命令完成不同的任务，如create、delete、edit、replace、apply等。

- Namespace和Pod是Kubernetes系统的基础资源类型。

第4章 管理Pod资源对象

Pod是Kubernetes系统的基础单元，是资源对象模型中可由用户创建或部署的最小组件，也是在Kubernetes系统上运行容器化应用的资源对象。其他的大多数资源对象都是用于支撑和扩展Pod对象功能的，例如，用于管控Pod运行的StatefulSet和Deployment等控制器对象，用于暴露Pod应用的Service和Ingress对象，为Pod提供存储的PersistentVolume存储资源对象等。这些资源对象大体可分为有限的几个类别，并且可基于资源清单作为资源配置文件进行陈述式或声明式管理。本章将描述这些类别，并详细介绍Pod资源的基础应用。

4.1 容器与Pod资源对象

现代的容器技术被设计用来运行单个进程（包括子进程）时，该进程在容器中PID名称空间中的进程号为1，可直接接收并处理信号，于是，在此进程终止时，容器即终止退出。若要在一个容器内运行多个进程，则需要为这些进程提供一个类似于Linux操作系统init进程的管控类进程，以树状结构完成多进程的生命周期管理，例如，崩溃后回收相应的系统资源等。单容器运行多进程时，通常还需要日志进程来管理这些进程的日志，例如，将它们分别保存于不同的目标日志文件等，否则用户就不得不手动来分拣日志信息。因此，绝大多数场景中都应该于一个容器中仅运行一个进程，它将日志信息直接输出至容器的标准输出，支持用户直接使用命令（`kubectl logs`）进行获取，这也是Docker及Kubernetes使用容器的标准方式。

不过，分别运行于各自容器的进程之间无法实现基于IPC的通信机制，此时，容器间的隔离机制对于依赖于此类通信方式的进程来说却又成了阻碍。Pod资源抽象正是用来解决此类问题的组件，前文已然多次提到，Pod对象是一组容器的集合，这些容器共享Network、UTS及IPC名称空间，因此具有相同的域名、主机名和网络接口，并可通过IPC直接通信。为一个Pod对象中的各容器提供网络名称空间等共享机制的是底层基础容器pause，图4-1所示为一个由三个容器组成的Pod资源，各容器共享Network、IPC和UTS名称空间，但分别拥有各自的MNT、USR和PID名称空间。需要特别强调的是，一个Pod对象中的多个容器必须运行于同一工作节点之上。

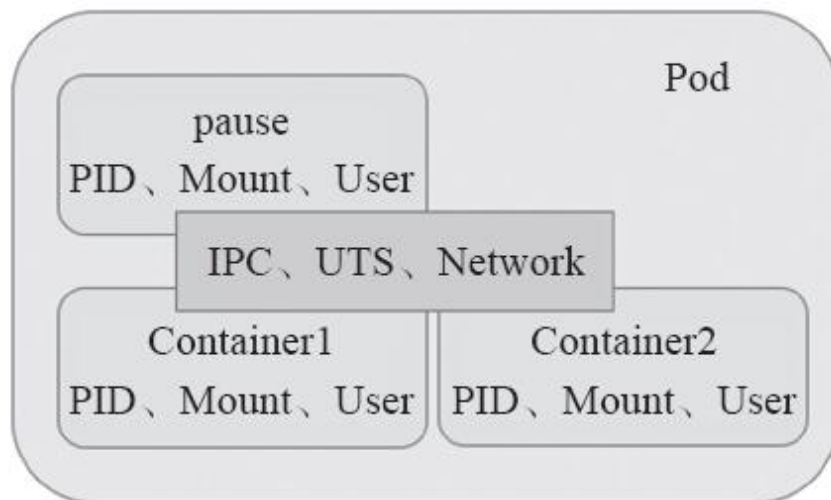


图4-1 Pod内的容器共享Network、IPC和UTS名称空间

尽管可以将Pod类比为物理机或VM，但一个Pod内通常仅应该运行一个应用，除非多个进程之间具有密切的关系。这也意味着，实践中应该将多个应用分别构建到多个而非单个Pod中，这样也更能符合容器的轻量化设计、运行之目的。

例如，一个有着前端（**application server**）和后端（**database server**）的应用，其前、后端应该分别组织于各自的Pod中，而非同一Pod的不同容器中。这样做的好处在于，多个Pod可被调度至多个不同的主机运行，提高了资源利用率。另外，Pod也是Kubernetes进行系统规模伸缩的基础单元，分别运行于不同Pod的多个应用可独立按需进行规模变动，这就增强了系统架构的灵活性。事实上，前、后端应用的规模需求通常不会相同，而且无状态应用（**application server**）的规模变动也比有状态应用（**database server**）容易得多，将它们组织于同一Pod中时将无法享受这种便利。

不过，有些场景要求必须于同一Pod中同时运行多个容器。此时，这些分布式应用（尤其是微服务架构中的多个服务）必须遵循某些最佳实践机制或基本准则。事实上，Kubernetes并非期望成为一个管理系统，而是一个支持这些最佳实践的向开发人员或管理人员提供更高级别服务的系统。分布式系统设计通常包含以下几种模型。

1) **Sidecar pattern**（边车模型或跨斗模型）：即为Pod的主应用容器提供协同的辅助应用容器，每个应用独立运行，最为典型的代表是将主应用容器中的日志使用agent收集至日志服务器中时，可以将agent运行为辅助应用容器，即sidecar。另一个典型的应用是为主应用容器中的database server启用本地缓存，如图4-2所示。

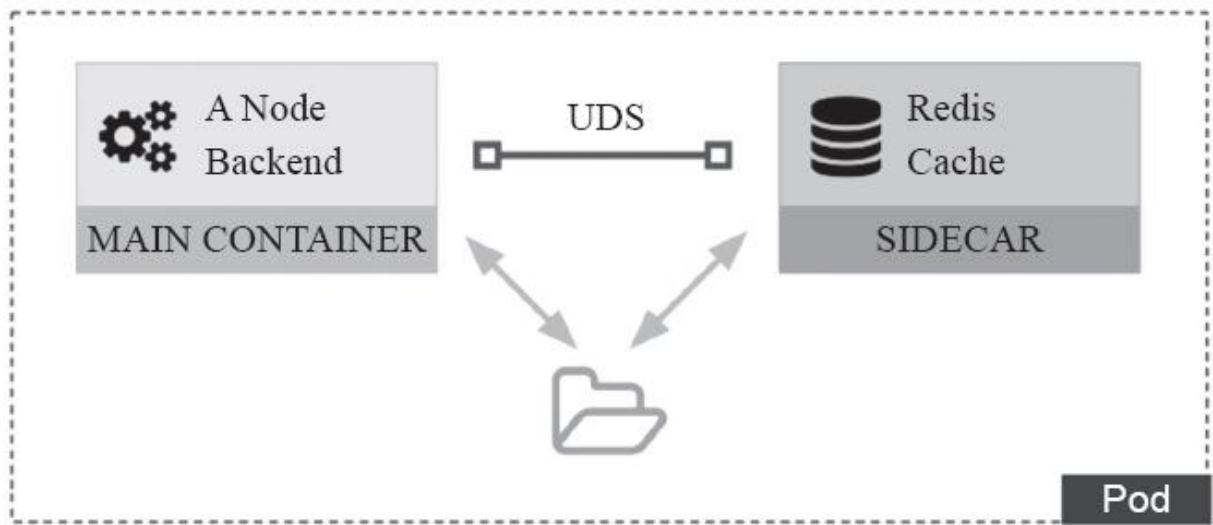


图4-2 Sidecar pattern

2) **Ambassador pattern** (大使模型)：即为远程服务创建一个本地代理，代理应用运行于容器中，主容器中的应用通过代理容器访问远程服务，如图4-3所示。一个典型的使用示例是主应用容器中的进程访问“一主多从”模型的远程Redis应用时，可在当前Pod容器中为Redis服务创建一个Ambassador container，主应用容器中的进程直接通过localhost接口访问Ambassador container即可。即便是Redis主从集群架构发生变动时，也仅需要将Ambassador container加以修改即可，主应用容器无须对此做出任何反应。

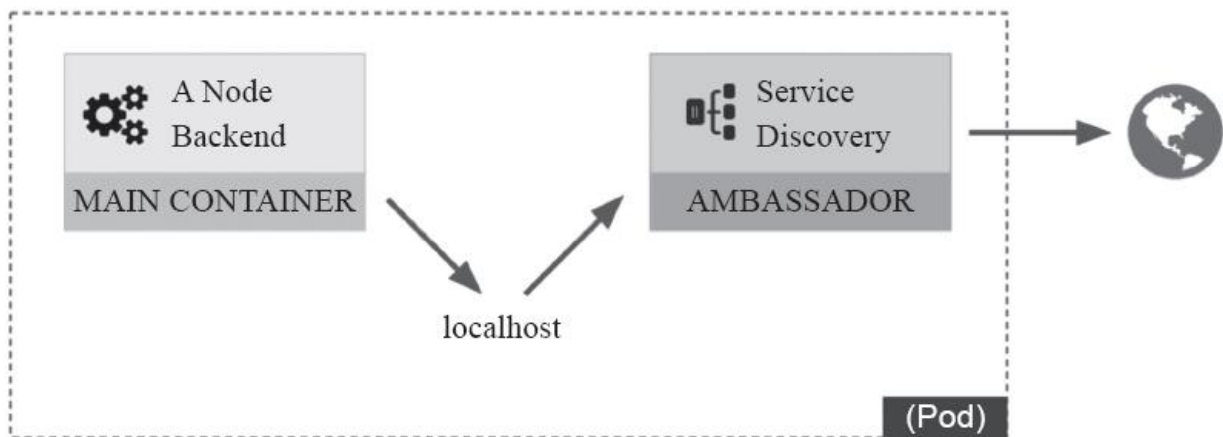


图4-3 Ambassador pattern

3) **Adapter pattern** (适配器模型)：此种模型一般用于将主应用容器中的内容进行标准化输出，例如，日志数据或指标数据的输出，这

有助于调用者统一接收数据的接口，如图4-4所示。另外，某应用滚动升级后的版本不兼容旧的版本时，其报告信息的格式也存在不兼容的可能性，使用Adapter container有助于避免那些调用此报告数据的应用发生错误。

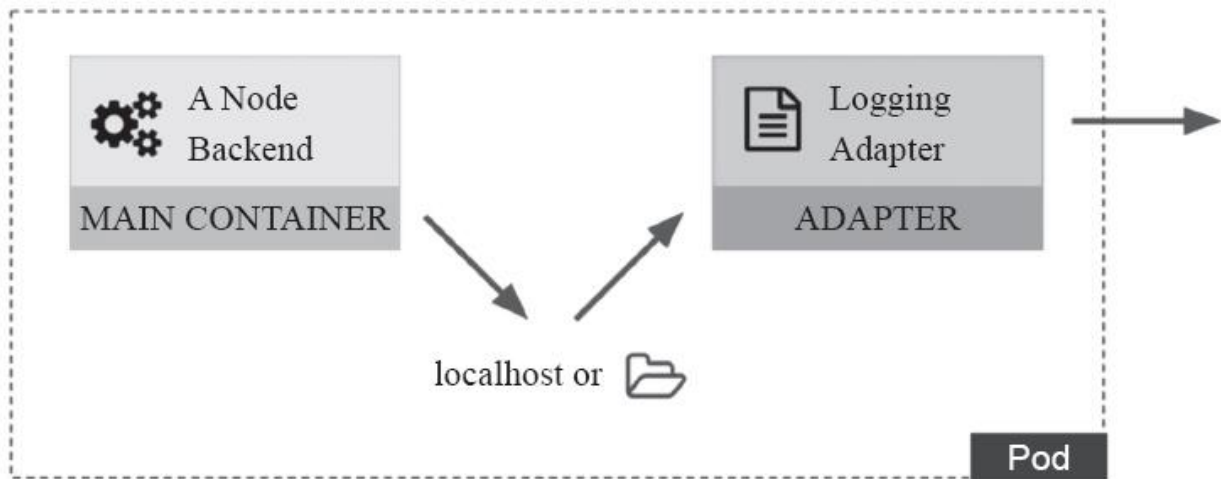


图4-4 Adapter pattern

Kubernetes系统的Pod资源对象用于运行单个容器化应用，此应用称为Pod对象的主容器（main container），同时Pod也能容纳多个容器，不过额外的容器一般工作为Sidecar模型，用于辅助主容器完成工作职能。

4.2 管理Pod对象的容器

一个Pod对象中至少要存在一个容器，因此，`containers`字段是定义Pod时其嵌套字段`Spec`中的必选项，用于为Pod指定要创建的容器列表。进行容器配置时，`name`为必选字段，用于指定容器名称，`image`字段是为可选，以方便更高级别的管理类资源（如`Deployment`）等能覆盖此字段，于是自主式的Pod并不可省略此字段。因此，定义一个容器的基础框架如下：

```
name: CONTAINER_NAME
image: IMAGE_FILE_NAME
```

此外，定义容器时还有一些其他常用的字段，例如，定义要暴露的端口、改变镜像运行的默认程序、传递环境变量、定义可用的系统资源配额等。

4.2.1 镜像及其获取策略

各工作节点负责运行Pod对象，而Pod的核心功用在于运行容器，因此工作节点上必须配置容器运行引擎，如Docker等。启动容器时，容器引擎将首先于本地查找指定的镜像文件，不存在的镜像则需要从指定的镜像仓库（Registry）下载至本地，如图4-5所示。

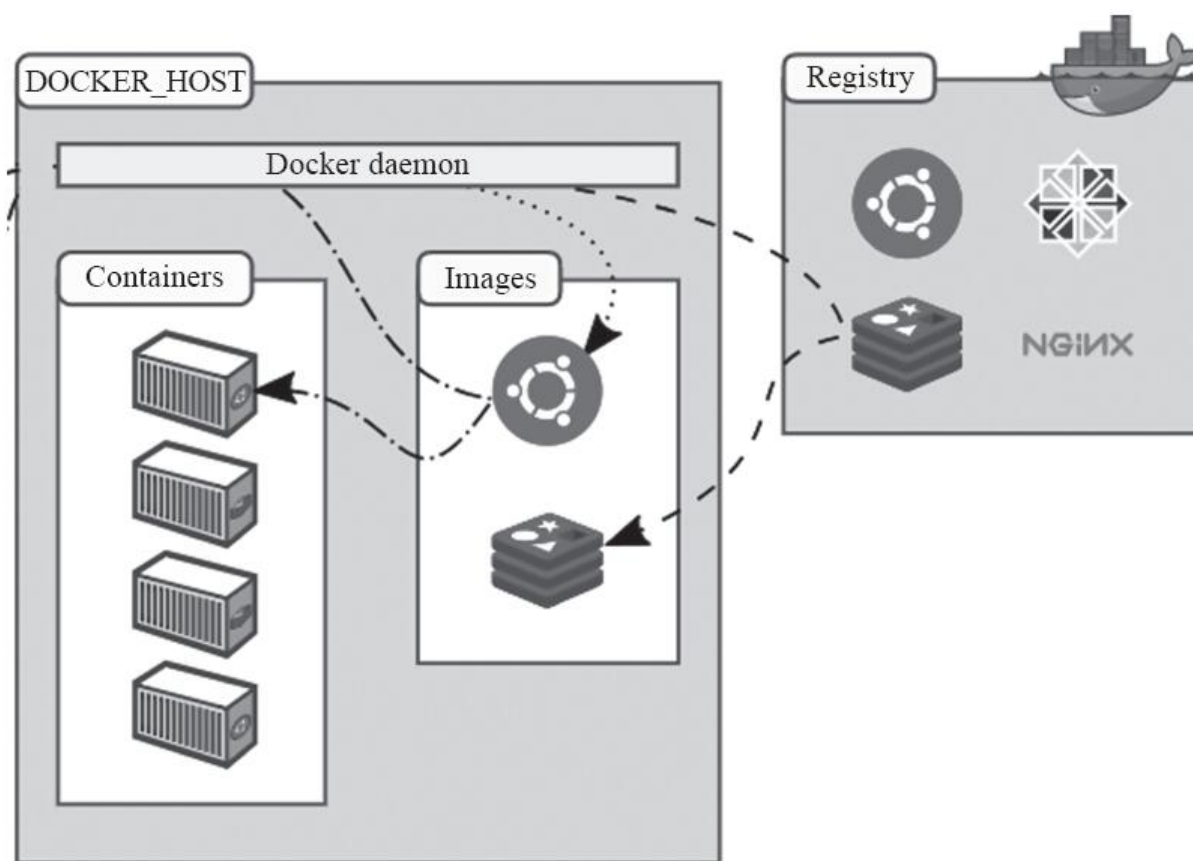


图4-5 Docker及其Registry

Kubernetes系统支持用户自定义镜像文件的获取策略，例如在网络资源较为紧张时可以禁止从仓库中获取镜像文件等。容器的“imagePullPolicy”字段用于为其指定镜像获取策略，它的可用值包括如下几个。

- Always: 镜像标签为“latest”或镜像不存在时总是从指定的仓库中获取镜像。

- IfNotPresent**: 仅当本地镜像缺失时方才从目标仓库下载镜像。

- Never**: 禁止从仓库下载镜像，即仅使用本地镜像。

下面的资源清单中的容器定义了如何使用**nginx: latest**镜像，其获取策略为**Always**，这意味着每次启动容器时，它都会到镜像仓库中获取最新版本的镜像文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
    imagePullPolicy: Always
```

对于标签为“**latest**”的镜像文件，其默认的镜像获取策略即为“**Always**”，而对于其他标签的镜像，其默认策略则为“**IfNotPresent**”。需要注意的是，使用私有仓库中的镜像时通常需要通过**Registry**服务器完成认证后才能进行。认证过程要么需要在相关节点上交互式执行**docker login**命令来进行，要么就是将认证信息定义为专有的**Secret**资源，并配置**Pod**通过“**imagePullSecretes**”字段调用此认证信息完成。后面8.5节会详细介绍此功能及其实现。

4.2.2 暴露端口

Docker的网络模型中，使用默认网络的容器化应用需通过NAT机制将其“暴露”（**expose**）到外部网络中才能被其他节点之上的容器客户端所访问。然而，Kubernetes系统的网络模型中，各Pod的IP地址处于同一网络平面，无论是否为容器指定了要暴露的端口，都不会影响集群中其他节点之上的Pod客户端对其进行访问，这就意味着，任何监听在非lo接口上的端口都可以通过Pod网络直接被请求。从这个角度来说，容器端口只是信息性数据，它只是为集群用户提供一个快速了解相关Pod对象的可访问端口的途径，而且显式指定容器端口，还能为其赋予一个名称以方便调用。

容器的ports字段的值是一个列表，由一到多个端口对象组成，它的常用嵌套字段包括如下几个。

- containerPort <integer>**: 必选字段，指定在Pod对象的IP地址上暴露的容器端口，有效范围为（0，65536）；使用时，应该总是指定容器应用正常监听着的端口。

- name <string>**: 当前端口的名称，必须符合IANA_SVC_NAME规范且在当前Pod内必须是唯一的；此端口名可被Service资源调用。

- protocol**: 端口相关的协议，其值仅可为TCP或UDP，默认为TCP。



提示 可以通过“`kubectl explain pods.spec.containers.ports`”获取ports对象可用的字段列表。

下面的资源配置清单示例（`pod-example-with-port.yaml`）中定义的pod-example指定了要暴露容器上TCP的80端口，并将之命名为http：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
```



```
- name: myapp
  image: ikubernetes/myapp:v1
  ports:
  - name: http
    containerPort: 80
    protocol: TCP
```

然而，Pod对象的IP地址仅在当前集群内可达，它们无法直接接收来自集群外部客户端的请求流量，尽管它们的服务可达性不受工作节点边界的约束，但依然受制于集群边界。一个简单的解决方案是通过其所在的工作节点的IP地址和端口将其暴露到集群外部，如图4-6所示。

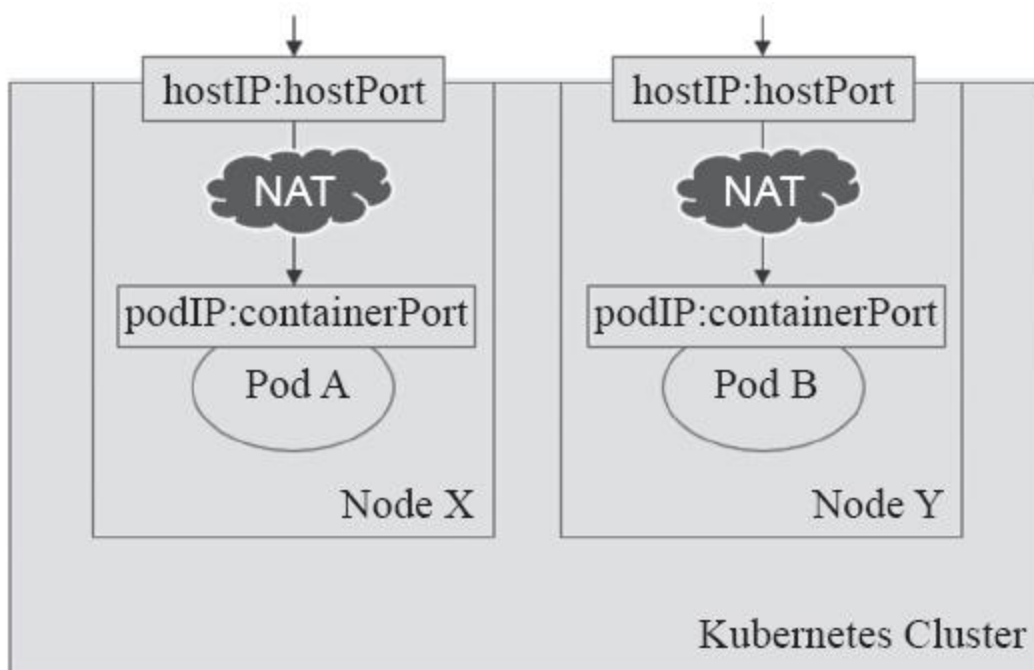


图4-6 通过hostIP和hostPort暴露容器服务

·hostPort <integer>: 主机端口，它将接收到的请求通过NAT机制转发至由containerPort字段指定的容器端口。

·hostIP <string>: 主机端口要绑定的主机IP，默认为0.0.0.0，即主机之上所有可用的IP地址；考虑到托管的Pod对象是由调度器调度运行的，工作节点的IP地址难以明确指定，因此此字段通常使用默认值。

需要注意的是，hostPort与NodePort类型的Service对象暴露端口的方式不同，NodePort是通过所有节点暴露容器服务，而hostPort则是经

由Pod对象所在节点的IP地址来进行。

4.2.3 自定义运行的容器化应用

由Docker镜像启动容器时运行的应用程序在相应的Dockerfile中由ENTRYPOINT指令进行定义，传递给程序的参数则通过CMD指令指定，ENTRYPOINT指令不存在时，CMD可用于同时指定程序及其参数。例如，在某工作节点上运行下面的命令获取ikubernetes/myapp:v1镜像中定义的CMD和ENTRYPOINT，命令如下：

```
~]$ docker inspect ikubernetes/myapp:v1 -f {{.Config.Cmd}}  
[nginx -g daemon off;]  
~]$ docker inspect ikubernetes/myapp:v1 -f {{.Config.Entrypoint}}  
[]
```

容器的command字段能够指定不同于镜像默认运行的应用程序，并且可以同时使用args字段进行参数传递，它们将覆盖镜像中的默认定义。不过，如果仅为容器定义了args字段，那么它将作为参数传递给镜像中默认指定运行的应用程序；如果仅为容器定义了command字段，那么它将覆盖镜像中定义的程序及参数，并以无参数方式运行应用程序。例如下面的资源清单文件将镜像ikubernetes/myapp: v1的默认应用程序修改为了“/bin/sh”，传递应用的参数修改为了“-c while true; do sleep 30; done”：

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: pod-with-custom-command  
spec:  
  containers:  
  - name: myapp  
    image: alpine:latest  
    command: ["/bin/sh"]  
    args: ["-c", "while true; do sleep 30; done"]
```

自定义args，也是向容器中的应用程序传递配置信息的常用方式之一，对于非云原生（cloud native）的应用程序，这几乎也是最简单的配置方式。另一个常用的方式是使用环境变量。

4.2.4 环境变量

非容器化的传统管理方式中，复杂应用程序的配置信息多数由配置文件进行指定，用户可借助于简单的文本编辑器完成配置管理。然而，对于容器隔离出的环境中的应用程序，用户就不得不穿透容器边界在容器内进行配置编辑并进行重载，这种方式复杂且低效。于是，由环境变量在容器启动时传递配置信息就成为一种备受青睐的方式。



注意 这种方式依赖于应用程序支持通过环境变量进行配置的能力，否则，用户在制作Docker镜像时需要通过entrypoint脚本完成环境变量到程序配置文件的同步。

向Pod对象中的容器环境变量传递数据的方法有两种：`env`和`envFrom`，这里重点介绍第一种方式，第二种方式将在介绍ConfigMap和Secret资源时进行说明。

通过环境变量配置容器化应用时，需要在容器配置段中嵌套使用`env`字段，它的值是一个由环境变量构成的列表。环境变量通常由`name`和`value`字段构成。

·`name<string>`：环境变量的名称，必选字段。

·`value<string>`：传递给环境变量的值，通过`$ (VAR_NAME)`引用，逃逸格式为“`$$ (VAR_NAME)`”，默认值为空。

下面配置清单中定义的Pod对象为其容器filebeat传递了两个环境变量，`REDIS_HOST`定义了filebeat收集的日志信息要发往的Redis主机地址，`LOG_LEVEL`则定义了filebeat的日志级别：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-env
spec:
  containers:
    - name: filebeat
      image: ikubernetes/filebeat:5.6.5-alpine
      env:
```

- name: REDIS_HOST
value: db.ilinux.io:6379
- name: LOG_LEVEL
value: info

这些环境变量可直接注入容器的shell环境中，无论它们是否真正被用到，使用`printenv`一类的命令都能在容器中获取到所有环境变量的列表。

4.2.5 共享节点的网络名称空间

同一个Pod对象的各容器均运行于一个独立的、隔离的Network名称空间中，共享同一个网络协议栈及相关的网络设备，如图4-7a所示。也有一些特殊的Pod对象需要运行于所在节点的名称空间中，执行系统级的管理任务，例如查看和操作节点的网络资源甚至是网络设备等，如图4-7b所示。

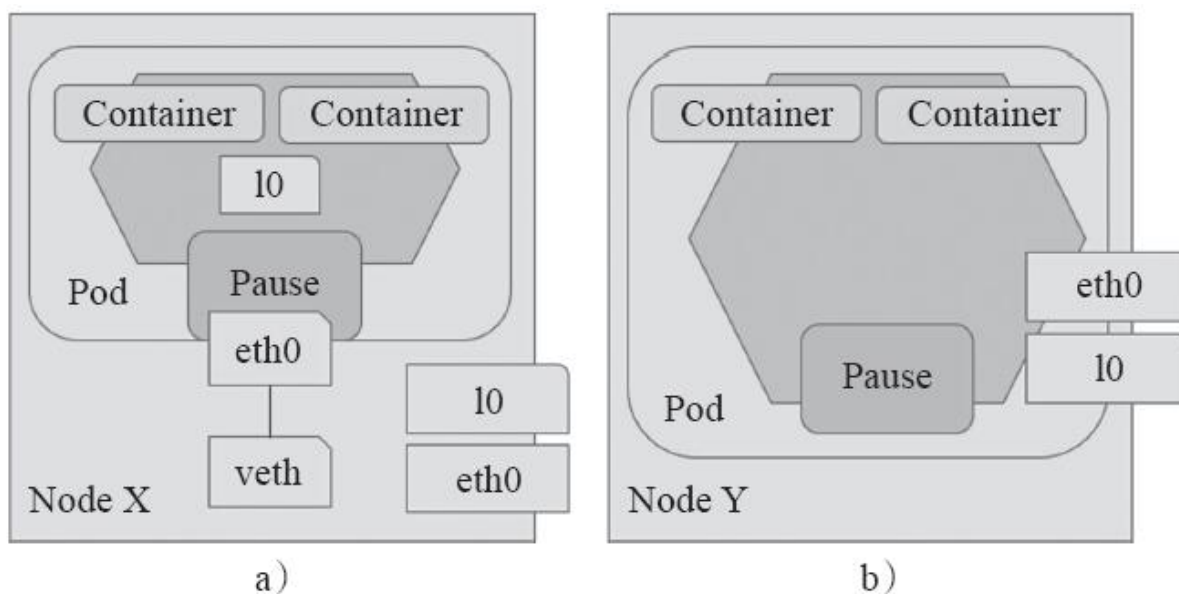


图4-7 Pod对象的网络名称空间

通常，以kubeadm部署的Kubernetes集群中的kube-apiserver、kube-controller-manager、kube-scheduler，以及kube-proxy和kube-flannel等通常都是第二种类型的Pod对象。事实上，仅需要设置spec.hostNetwork的属性为true即可创建共享节点网络名称空间的Pod对象，如下面的配置清单所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-use-hostnetwork
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
  hostNetwork: true
```

将上面的配置清单保存于配置文件中，如`pod-use-hostnetwork.yaml`，将其创建于集群上，并查看其网络接口的相关属性信息以验证它是否能共享使用工作节点的网络名称空间：

```
~]$ kubectl apply -f pod-use-hostnetwork.yaml
~]$ kubectl exec -it pod-use-hostnetwork -- sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:16:3E:08:1B:F4
          inet addr:172.16.0.68  Bcast:172.31.143.255  Mask:255.255.240.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
...
```

如上述命令的结果显示所示，它打印出的是工作节点的网络设备及其相关的接口信息。这就意味着，**Pod**对象中运行的容器化应用也将监听于其所在的工作节点的IP地址之上，这可以通过直接向`node03.ilinux.io`节点发起请求来验证：

```
~]$ curl node03.ilinux.io
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

另外，在**Pod**对象中时还可以分别使用`spec.hostPID`和`spec.hostIPC`来共享工作节点的PID和IPC名称空间。

4.2.6 设置Pod对象的安全上下文

Pod对象的安全上下文用于设定Pod或容器的权限和访问控制功能，其支持设置的常用属性包括以下几个方面。

- 基于用户ID（UID）和组ID（GID）控制访问对象（如文件）时的权限。
- 以特权或非特权的方式运行。
- 通过Linux Capabilities为其提供部分特权。
- 基于Seccomp过滤进程的系统调用。
- 基于SELinux的安全标签。
- 是否能够进行权限升级。

Pod对象的安全上下文定义在spec.securityContext字段中，而容器的安全上下文则定义在spec.containers[].securityContext字段中，且二者可嵌套使用的字段还有所不同。下面的配置清单示例为busybox容器定义了安全上下文，它以uid为1000的非特权用户运行容器，并禁止权限升级：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-securitycontext
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c", "sleep 86400"]
    securityContext:
      runAsNonRoot: true
      runAsUser: 1000
      allowPrivilegeEscalation: false
```

将上面的配置清单保存于配置文件（如pod-with-securitycontext.yaml文件）中，而后创建于集群中即可验证容器进程的

运行者身份:

```
$ kubectl apply -f pod-with-securitycontext.yaml
$ kubectl exec pod-with-securitycontext -- ps aux
PID    USER      TIME  COMMAND
   1   1000        0:00 sleep 86400
  14   1000        0:00 ps aux
```

另外，可设置的安全上下文属性还有`fsGroup`、`seLinuxOptions`、`supplementalGroups`、`sysctls`、`capabilities`和`privileged`等，且Pod和容器各自支持的字段也有所不同，感兴趣的读者可按需对各属性进行测试。

4.3 标签与标签选择器

实践中，随着同类型资源对象的数量越来越多，分类管理也越来越有必要：基于简单且直接的标准将资源对象划分为多个较小的分组，无论是对开发人员还是对系统工程师来说，都能提升管理效率，这也正是Kubernetes标签（Label）的核心功能之一。对于附带标签的资源对象，可使用标签选择器（Label Selector）挑选出符合过滤条件的资源以完成所需要的操作，如关联、查看和删除等。

4.3.1 标签概述

标签是Kubernetes极具特色的功能之一，它能够附加于Kubernetes的任何资源对象之上。简单来说，标签就是“键值”类型的数据，它们可于资源创建时直接指定，也可随时按需添加于活动对象中，而后即可由标签选择器进行匹配度检查从而完成资源挑选。一个对象可拥有不止一个标签，而同一个标签也可被添加至多个资源之上。

实践中，可以为资源附加多个不同纬度的标签以实现灵活的资源分组管理功能，例如，版本标签、环境标签、分层架构标签等，用于交叉标识同一个资源所属的不同版本、环境及架构层级等，如图4-8所示。下面是较为常用的标签。

·版本标

签: "release": "stable", "release": "canary", "release": "beta"。

·环境标

签: "environment": "dev", "environment": "qa", "environment": "production"。

·应用标

签: "app": "ui", "app": "as", "app": "pc", "app": "sc"。

·架构层级标

签: "tier": "frontend", "tier": "backend", "tier": "cache"。

·分区标签: "partition": "customerA", "partition": "customerB"。

·品控级别标签: "track": "daily", "track": "weekly"。

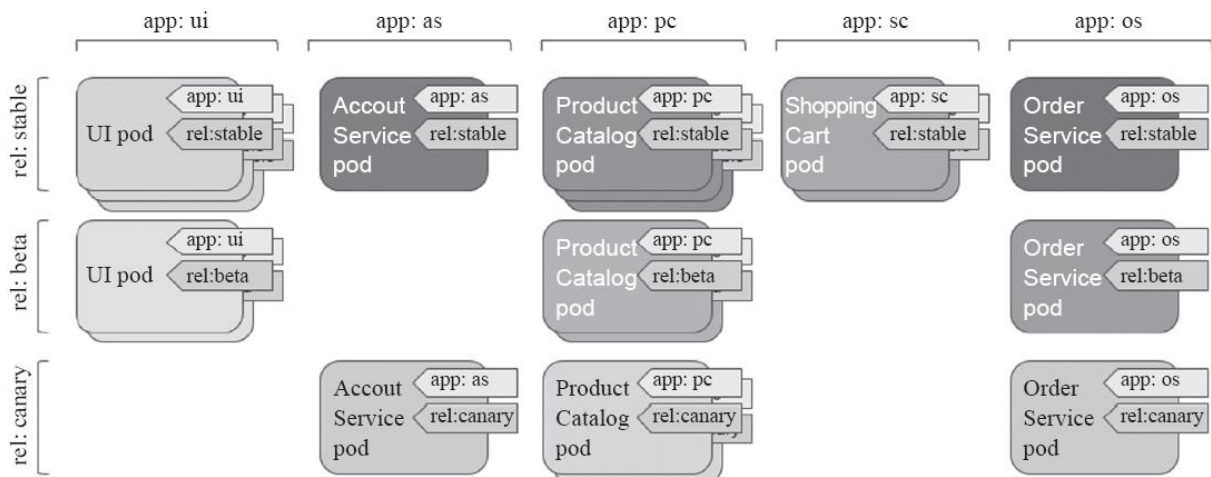


图4-8 多维度标签使用示例（图片来源：《Kubernetes in action》）

标签中的键名称通常由键前缀和键名组成，其中键前缀可选，其格式形如“KEY_PREFIX/KEY_NAME”。键名至多能使用63个字符，可使用字母、数字、连接号（-）、下划线（_）、点号（.）等字符，并且只能以字母或数字开头。键前缀必须为DNS子域名格式，且不能超过253个字符。省略键前缀时，键将被视为用户的私有数据，不过由Kubernetes系统组件或第三方组件自动为用户资源添加的键必须使用键前缀，而“kubernetes.io/”前缀则预留给Kubernetes的核心组件使用。

标签中的键值必须不能多于63个字符，它要么为空，要么是以字母或数字开头及结尾，且中间仅使用了字母、数字、连接号（-）、下划线（_）或点号（.）等字符的数据。



提示 实践中，建议键名及键值能做到“见名知义”，且尽可能保持简单。

4.3.2 管理资源标签

创建资源时，可直接在其metadata中嵌套使用“labels”字段以定义要附加的标签项。例如，下面的Pod资源清单文件示例pod-with-labels.yaml中使用了两个标签env=qa和tier=frontend：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    env: qa
    tier: frontend
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
```

基于此资源清单创建出定义的Pod对象之后，即可在“kubectl get pods”命令中使用“--show-labels”选项，以额外显示对象的标签信息：

```
~]$ kubectl apply -f pod-with-labels.yaml
pod "pod-with-labels" created
~]$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
pod-example	1/1	Running	0	3h	<none>
pod-with-labels	1/1	Running	0	13s	env=qa,tier=frontend

标签较多时，在“kubectl get pods”命令上使用“-L key1, key2, ...”选项可指定显示有着特定键的标签信息。例如，仅显示各pods之上的以env和tier为键名的标签：

```
~]$ kubectl get pods -L env,tier
```

NAME	READY	STATUS	RESTARTS	AGE	ENV	TIER
pod-example	1/1	Running	2	3h		
pod-with-labels	1/1	Running	0	5m	qa	frontend

“kubectl label”命令可以直接管理活动对象的标签，以按需进行添加或修改等操作。例如，为pod-example添加env=production标签：

```
~]$ kubectl label pods/pod-example env=production  
pod "pod-example" labeled
```

不过，对于已经附带了指定键名的标签，使用“`kubectl label`”为其设定新的键值时需要为命令同时使用“`--overwrite`”命令以强制覆盖原有的键值。例如，将`pod-with-labels`的`env`的值修改为“`testing`”：

```
~]$ kubectl label pods/pod-with-labels env=testing --overwrite  
pod "pod-with-labels" labeled
```

用户若期望对某标签之下的资源集合执行某类操作，例如，查看或删除等，则需要先使用“标签选择器”挑选出满足条件的资源对象。

4.3.3 标签选择器

标签选择器用于表达标签的查询条件或选择标准，Kubernetes API 目前支持两个选择器：基于等值关系（equality-based）以及基于集合关系（set-based）。例如，`env=production`和`env!=qa`是基于等值关系的选择器，而`tier in (frontend, backend)`则是基于集合关系的选择器。另外，使用标签选择器时还将遵循以下逻辑。

- 1) 同时指定的多个选择器之间的逻辑关系为“与”操作。
- 2) 使用空值的标签选择器意味着每个资源对象都将被选中。
- 3) 空的标签选择器将无法选出任何资源。

基于等值关系的标签选择器的可用操作符有“=”“==”和“!=”三种，其中前两个意义相同，都表示“等值”关系，最后一个表示“不等”关系。“`kubectl get`”命令的“-l”选项能够指定使用标签选择器，例如，显示键名`env`的值不为`qa`的所有Pod对象：

```
~]$ kubectl get pods -l "env!=qa" -L env
```

NAME	READY	STATUS	RESTARTS	AGE	ENV
pod-example	1/1	Running	2	4h	production
pod-with-labels	1/1	Running	0	40m	testing

再例如，显示标签键名`env`的值不为`qa`，且标签键名`tier`的值为`frontend`的所有Pod对象：

```
~]$ kubectl get pods -l "env!=qa,tier=frontend" -L env,tier
```

NAME	READY	STATUS	RESTARTS	AGE	ENV	TIER
pod-with-labels	1/1	Running	0	43m	testing	frontend

基于集合关系的标签选择器支持`in`、`notin`和`exists`三种操作符，它们的使用格式及意义具体如下。

·**KEY in (VALUE1, VALUE2, ...)**：指定的键名的值存在于给定的列表中即满足条件。

·**KEY notin** (VALUE1, VALUE2, ...)：指定的键名的值不存在于给定的列表中即满足条件。

·**KEY**：所有存在此键名标签的资源。

·**! KEY**：所有不存在此键名标签的资源。

例如，显示标签键名env的值为production或dev的所有Pod对象：

```
~]$ kubectl get pods -l "env in (production,dev)" -L env
```

NAME	READY	STATUS	RESTARTS	AGE	ENV
pod-example	1/1	Running	2	4h	production

再如，列出标签键名env的值为production或dev，且不存在键名为tier的标签的所有Pod对象：

```
~]$ kubectl get pods -l 'env in (production,dev),!tier' -L env,tier
```

NAME	READY	STATUS	RESTARTS	AGE	ENV	TIER
pod-example	1/1	Running	2		4h	production



注意 为了避免shell解释器解析叹号（!），必须要为此类表达式使用单引号。

此外，Kubernetes的诸多资源对象必须以标签选择器的方式关联到Pod资源对象，例如Service、Deployment和ReplicaSet类型的资源等，它们在spec字段中嵌套使用嵌套的“selector”字段，通过“matchLabels”来指定标签选择器，有的甚至还支持使用“matchExpressions”构造复杂的标签选择机制。

·**matchLabels**：通过直接给定键值对来指定标签选择器。

·**matchExpressions**：基于表达式指定的标签选择器列表，每个选择器都形如“{key: KEY_NAME, operator: OPERATOR, values: [VALUE1, VALUE2, ...]}”，选择器列表间为“逻辑与”关系；使用In或NotIn操作符时，其values不强制要求为非空的字符串列表，而使用Exists或DoesNotExist时，其values必须为空。

下面所示的资源清单片段是一个示例，它同时定义了两类标签选择器：

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: Exists, values: {}}
```

标签赋予了Kubernetes灵活操作资源对象的能力，它也是Service和Deployment等核心资源类型得以实现的基本前提。

4.3.4 Pod节点选择器nodeSelector

Pod节点选择器是标签及标签选择器的一种应用，它能够让Pod对象基于集群中工作节点的标签来挑选倾向运行的目标节点。

Kubernetes的kube-scheduler守护进程负责在各工作节点中基于系统资源的可用性等标签挑选一个来运行待创建的Pod对象，默认的调度器是default-scheduler。Kubernetes可将所有工作节点上的各系统资源抽象成资源池统一分配使用，因此用户无须关心Pod对象的具体运行位置也能良好工作。不过，事情总有例外，比如仅有部分节点拥有被Pod对象依赖到的特殊硬件设备的情况，如GPU和SSD等。即便如此，用户也不应该静态指定Pod对象的运行位置，而是让scheduler基于标签和标签选择器为Pod挑选匹配的工作节点。

Pod对象的spec.nodeSelector可用于定义节点标签选择器，用户事先为特定部分的Node资源对象设定好标签，而后配置Pod对象通过节点标签选择器进行匹配检测，从而完成节点亲和性调度。

为Node资源对象附加标签的方法同Pod资源，使用“kubectl label nodes/NODE”命令即可。例如，可为node01.ilinux.io和node03.ilinux.io节点设置“disktype=ssd”标签以标识其拥有SSD设备：

```
~]$ kubectl label nodes node01.ilinux.io disktype=ssd
node "node01.ilinux.io" labeled
~]$ kubectl label nodes node03.ilinux.io disktype=ssd
node "node03.ilinux.io" labeled
```

查看具有键名SSD的标签的Node资源：

```
~]$ kubectl get nodes -l 'disktype' -L disktype
```

NAME	STATUS	ROLES	AGE	VERSION	DISKTYPE
node01.ilinux.io	Ready	<none>	15d	v1.12.1	ssd
node03.ilinux.io	Ready	<none>	15d	v1.12.1	ssd

如果某Pod资源需要调度至这些具有SSD设备的节点之上，那么只需要为其使用spec.nodeSelector标签选择器即可，例如下面的资源清单

文件pod-with-nodeselector.yaml示例:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-nodeselector
  labels:
    env: testing
spec:
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
  nodeSelector:
    disktype: ssd
```

将如上资源清单中定义的Pod资源创建于集群中，通过查看其运行的结点即可判定调度效果。

另外，动手测试和查看过节点标签的读者或许已经注意到了，集群中的每个节点默认已经附带了多个标签，如kubernetes.io/hostname、beta.kubernetes.io/os和beta.kubernetes.io/arch等。这些标签也可以直接由nodeSelector使用，尤其是希望将Pod调度至某特定节点时，可以使用kubernetes.io/hostname直接绑定至相应的主机即可。不过，这种绑定至特定主机的需求还有一种更为简单的实现方式，即使用spec.nodeName字段直接指定目标节点。

4.4 资源注解

除了标签（**label**）之外，**Pod**与其他各种资源还能使用资源注解（**annotation**）。与标签类似，注解也是“键值”类型的数据，不过它不能用于标签及挑选**Kubernetes**对象，仅可用于为资源提供“元数据”信息。另外，注解中的元数据不受字符数量的限制，它可大可小，可以为结构化或非结构化形式，也支持使用在标签中禁止使用的其他字符。

资源注解可由用户手动添加，也可由工具程序自动附加并使用它们。在**Kubernetes**的新版本中（**alpha**或**beta**阶段）为某资源引入新字段时，常以注解的方式提供，以避免其增删等变动对用户带来困扰，一旦确定支持使用它们，这些新增字段就将再引入到资源中并淘汰相关的注解。另外，为资源添加注解也可让其他用户快速了解资源的相关信息，例如其创建者的身份等。以下为常用的场景案例。

- 由声明式配置层（如**apply**命令）管理的字段：将这些字段定义为注解有助于识别由服务器或客户端设定的默认值、系统自动生成的字段以及由自动伸缩系统生成的字段。

- 构建、发行或镜像相关的信息，例如，时间戳、发行ID、Git分支、PR号码、镜像哈希及仓库地址等。

- 指向日志、监控、分析或审计仓库的指针。

- 由客户端库或工具程序生成的用于调试目的的信息：如名称、版本、构建信息等。

- 用户或工具程序的来源地信息，例如，来自其他生态系统组件的相关对象的url。

- 轻量化滚动升级工具的元数据，如**config**及**checkpoints**。

- 相关人员的电话号码等联系信息，或者指向类似信息的可寻址的目录条目，如网站站点。

4.4.1 查看资源注解

“`kubectl get-o yaml`”和“`kubectl describe`”命令均能显示资源的注解信息。例如下面的命令显示的pod-example的注解信息：

```
~]$ kubectl describe pods pod-example
Name:          pod-example
Namespace:     default
Node:          node02.ilinux.io/172.16.0.67
Start Time:    Mon, 26 Feb 2018 18:40:53 +0800
Labels:        env=production
Annotations:   kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v
              1","kind":"Pod","metadata":{"annotations":{},"name":"pod-example","namespa
              ce":"default"},"spec":{"containers":[{"image":"ikubernetes/m..
Status:        Running
.....
```

pod-example此前由声明式配置命令`apply`创建，因此它在注解中保存了如上的相关信息以便在下次资源变动时进行版本对比。

4.4.2 管理资源注解

`annotations`可在资源创建时使用“`metadata.annotations`”字段指定，也可随时按需在活动的资源上使用“`kubectl annotate`”命令进行附加。例如，为`pod-example`重新进行注解：

```
~]$ kubectl annotate pods pod-example ilinux.io/created-by="cluster admin"
pod "pod-example" annotated
```

查看生成的注解信息：

```
~]$ kubectl describe pods pod-example | grep "Annotations"
Annotations:  ilinux.io/created-by=cluster admin
```

如果需要在资源创建时的清单中指定，那么使用类似如下的方式即可：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  annotations:
    ilinux.io/created-by: cluster admin
spec:
.....
```

4.5 Pod对象的生命周期

Pod对象自从其创建开始至其终止退出的时间范围称为其生命周期。在这段时间中，Pod会处于多种不同的状态，并执行一些操作；其中，创建主容器（**main container**）为必需的操作，其他可选的操作还包括运行初始化容器（**init container**）、容器启动后钩子（**post start hook**）、容器的存活性探测（**liveness probe**）、就绪性探测（**readiness probe**）以及容器终止前钩子（**pre stop hook**）等，这些操作是否执行则取决于Pod的定义，如图4-9所示。

4.5.1 Pod的相位

无论是类似前面几节中的由用户手动创建，还是通过Deployment等控制器创建，Pod对象总是应该处于其生命进程中以下几个相位（phase）之一。

- Pending**: API Server创建了Pod资源对象并已存入etcd中，但它尚未被调度完成，或者仍处于从仓库下载镜像的过程中。

- Running**: Pod已经被调度至某节点，并且所有容器都已经被kubelet创建完成。

- Succeeded**: Pod中的所有容器都已经成功终止并且不会被重启。

- Failed**: 所有容器都已经终止，但至少有一个容器终止失败，即容器返回了非0值的退出状态或已经被系统终止。

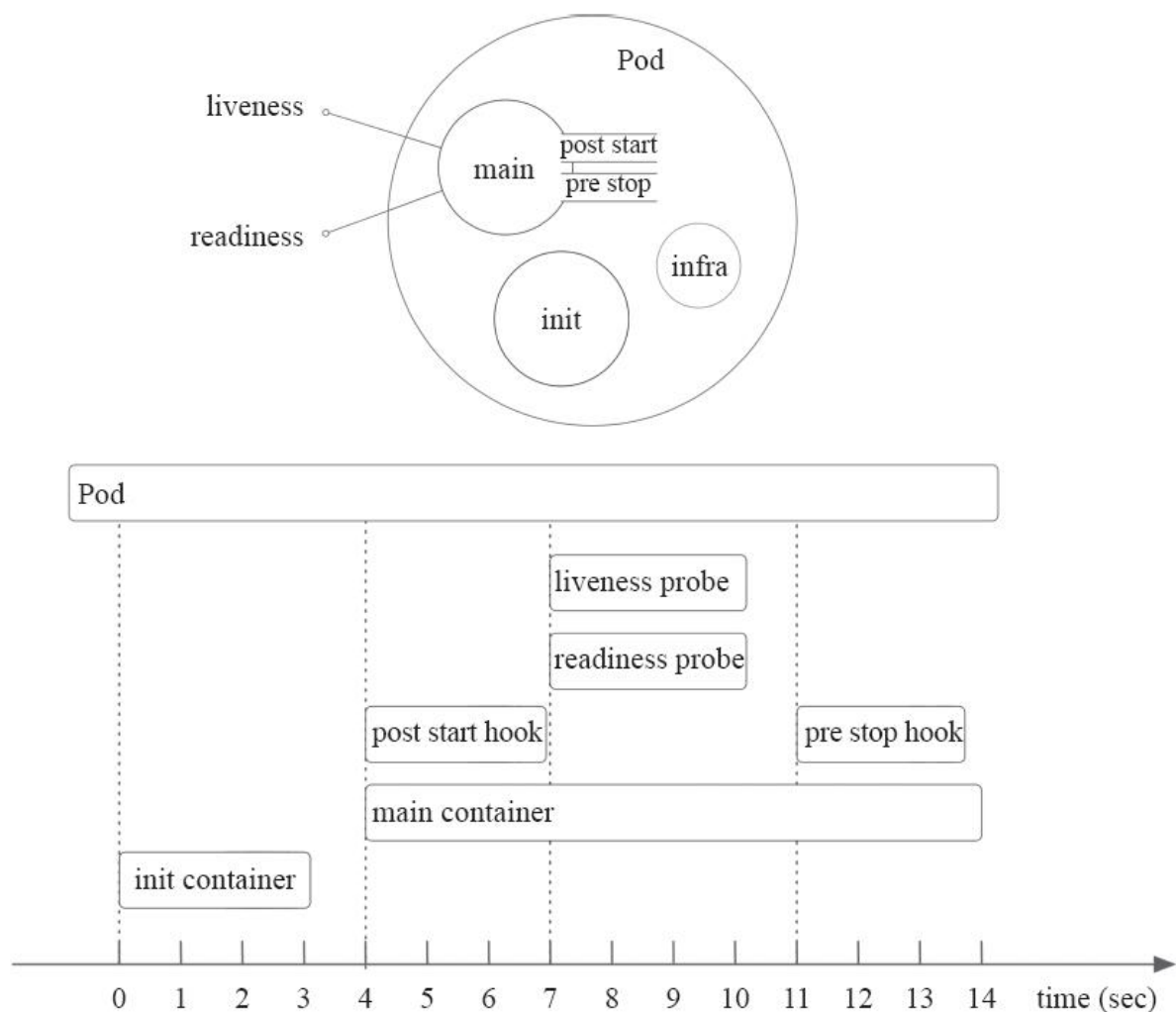


图4-9 Pod的生命周期（图片来源：<https://blog.openshift.com>）

·Unknown: API Server无法正常获取到Pod对象的状态信息，通常是由于其无法与所在工作节点的kubelet通信所致。

Pod的相位是在其生命周期中的宏观概述，而非对容器或Pod对象的综合汇总，而且相位的数量和含义被严格界定，它仅包含上面列举的相位值。

4.5.2 Pod的创建过程

Pod是Kubernetes的基础单元，理解它的创建过程对于了解系统运作大有裨益。图4-10描述了一个Pod资源对象的典型创建过程。

- 1) 用户通过kubectl或其他API客户端提交Pod Spec给API Server。
- 2) API Server尝试着将Pod对象的相关信息存入etcd中，待写入操作执行完成，API Server即会返回确认信息至客户端。
- 3) API Server开始反映etcd中的状态变化。
- 4) 所有的Kubernetes组件均使用“watch”机制来跟踪检查API Server上的相关的变动。
- 5) kube-scheduler（调度器）通过其“watcher”觉察到API Server创建了新的Pod对象但尚未绑定至任何工作节点。
- 6) kube-scheduler为Pod对象挑选一个工作节点并将结果信息更新至API Server。

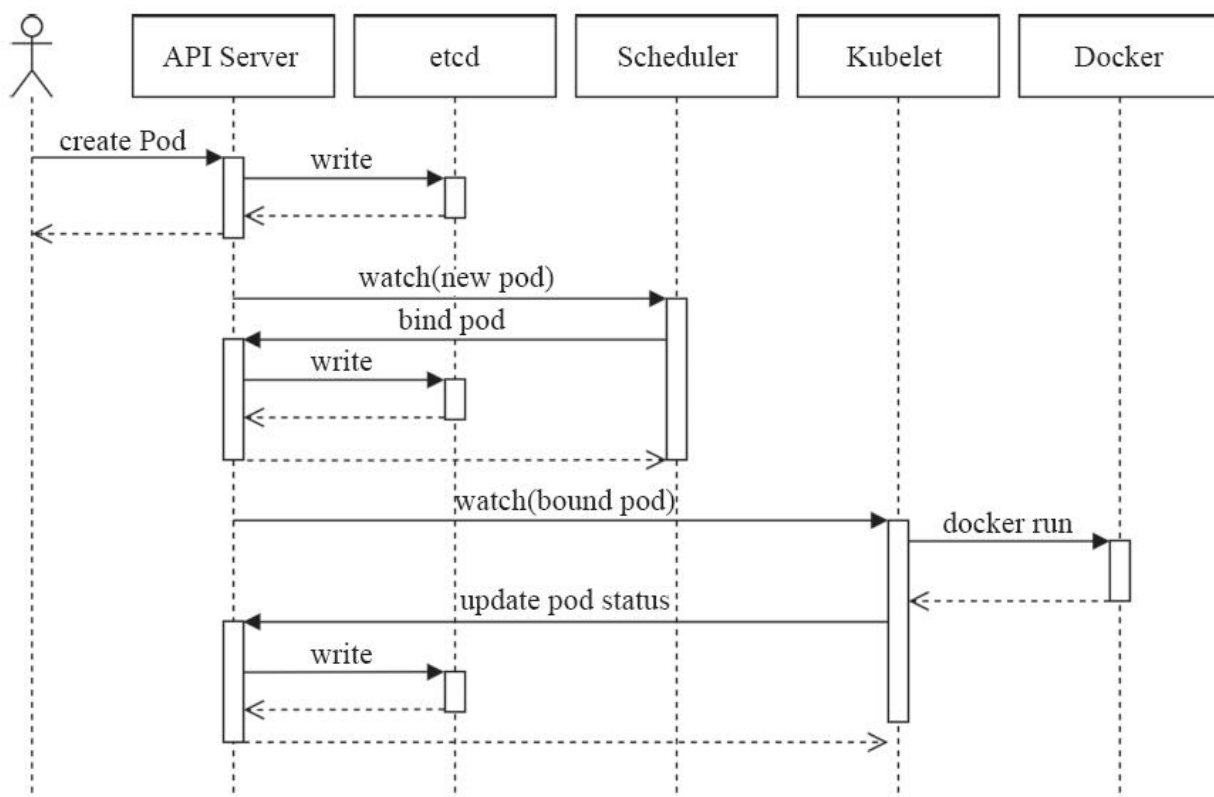


图4-10 Pod资源对象创建过程（图片来源：<https://blog.heptio.com/>）

7) 调度结果信息由API Server更新至etcd存储系统，而且API Server也开始反映此Pod对象的调度结果。

8) Pod被调度到的目标工作节点上的kubelet尝试在当前节点上调用Docker启动容器，并将容器的结果状态回送至API Server。

9) API Server将Pod状态信息存入etcd系统中。

10) 在etcd确认写入操作成功完成后，API Server将确认信息发送至相关的kubelet，事件将通过它被接受。

4.5.3 Pod生命周期中的重要行为

除了创建应用容器（主容器及其辅助容器）之外，用户还可以为Pod对象定义其生命周期中的多种行为，如初始化容器、存活性探测及就绪性探测等。

1.初始化容器

初始化容器（**init container**）即应用程序的主容器启动之前要运行的容器，常用于为主容器执行一些预置操作，它们具有两种典型特征。

- 1) 初始化容器必须运行完成直至结束，若某初始化容器运行失败，那么Kubernetes需要重启它直到成功完成。
- 2) 每个初始化容器都必须按定义的顺序串行运行。



注意 如果Pod的spec.restartPolicy字段值为“Never”，那么运行失败的初始化容器不会被重启。

有不少场景都需要在应用容器启动之前进行部分初始化操作，例如，等待其他关联组件服务可用、基于环境变量或配置模板为应用程序生成配置文件、从配置中心获取配置等。初始化容器的典型应用需求具体包含如下几种。

- 1) 用于运行特定的工具程序，出于安全等方面的原因，这些程序不适于包含在主容器镜像中。
- 2) 提供主容器镜像中不具备的工具程序或自定义代码。
- 3) 为容器镜像的构建和部署人员提供了分离、独立工作的途径，使得他们不必协同起来制作单个镜像文件。
- 4) 初始化容器和主容器处于不同的文件系统视图中，因此可以分别安全地使用敏感数据，例如Secrets资源。

5) 初始化容器要先于应用容器串行启动并运行完成，因此可用于延后应用容器的启动直至其依赖的条件得到满足。

Pod资源的“spec.initContainers”字段以列表的形式定义可用的初始容器，其嵌套可用字段类似于“spec.containers”。下面的资源清单仅是一个初始化容器的使用示例，读者可自行创建并观察初始化容器的相关状态：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: ikubernetes/myapp:v1
  initContainers:
  - name: init-something
    image: busybox
    command: ['sh', '-c', 'sleep 10']
```

2.生命周期钩子函数

生命周期钩子函数（lifecycle hook）是编程语言（如Angular）中常用的生命周期管理的组件，它实现了程序运行周期中的关键时刻的可见性，并赋予用户为此采取某种行动的能力。类似地，容器生命周期钩子使它能够感知其自身生命周期管理中的事件，并在相应的时刻到来时运行由用户指定的处理程序代码。Kubernetes为容器提供了两种生命周期钩子。

- postStart**: 于容器创建完成之后立即运行的钩子处理器（handler），不过Kubernetes无法确保它一定会于容器中的ENTRYPOINT之前运行。

- preStop**: 于容器终止操作之前立即运行的钩子处理器，它以同步的方式调用，因此在其完成之前会阻塞删除容器的操作的调用。

钩子处理器的实现方式有“Exec”和“HTTP”两种，前一种在钩子事件触发时直接在当前容器中运行由用户定义的命令，后一种则是在当前容器中向某URL发起HTTP请求。

`postStart`和`preStop`处理器定义在容器的`spec.lifecycle`嵌套字段中，其使用方法如下面的资源清单所示，读者可自行创建相关的Pod资源对象，并验证其执行结果：

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: ikubernetes/myapp:v1
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo 'lifecycle hooks handler' > /usr/share/nginx/html/test.html"]
```

3.容器探测

容器探测（**container probe**）是Pod对象生命周期中的一项重要的重要的日常任务，它是kublet对容器周期性执行的健康状态诊断，诊断操作由容器的处理器（**handler**）进行定义。Kubernetes支持三种处理器用于Pod探测。

- ExecAction**：在容器中执行一个命令，并根据其返回的状态码进行诊断的操作称为**Exec**探测，状态码为0表示成功，否则即为不健康状态。

- TCPSocketAction**：通过与容器的某TCP端口尝试建立连接进行诊断，端口能够成功打开即为正常，否则为不健康状态。

- HTTPGetAction**：通过向容器IP地址的某指定端口的指定path发起HTTP GET请求进行诊断，响应码为2xx或3xx时即为成功，否则为失败。

任何一种探测方式都可能存在三种结果：“**Success**”（成功）、“**Failure**”（失败）或“**Unknown**”（未知），只有第一种结果表示成功通过检测。

- kublet可在活动容器上执行两种类型的检测：存活性检测（**livenessProbe**）和就绪性检测（**readinessProbe**）。

·存活性检测：用于判定容器是否处于“运行”（Running）状态；一旦此类检测未通过，`kubelet`将杀死容器并根据其`restartPolicy`决定是否将其重启；未定义存活性检测的容器的默认状态为“Success”。就绪性检测：用于判断容器是否准备就绪并可对外提供服务；未通过检测的容器意味着其尚未准备就绪，端点控制器（如Service对象）会将其IP从所有匹配到此Pod对象的Service对象的端点列表中移除；检测通过之后，会再次将其IP添加至端点列表中。



提示 存活性检测和就绪性检测相关的话题在后文的章节中还会有进一步的介绍。

4.5.4 容器的重启策略

容器程序发生崩溃或容器申请超出限制的资源等原因都可能会导致Pod对象的终止，此时是否应该重建该Pod对象则取决于其重启策略（restartPolicy）属性的定义。

- 1) Always: 但凡Pod对象终止就将其重启，此为默认设定。
- 2) OnFailure: 仅在Pod对象出现错误时方才将其重启。
- 3) Never: 从不重启。

需要注意的是，restartPolicy适用于Pod对象中的所有容器，而且它仅用于控制在同一节点上重新启动Pod对象的相关容器。首次需要重启的容器，将在其需要时立即进行重启，随后再次需要重启的操作将由kubelet延迟一段时间后进行，且反复的重启操作的延迟时长依次为10秒、20秒、40秒、80秒、160秒和300秒，300秒是最大延迟时长。事实上，一旦绑定到一个节点，Pod对象将永远不会被重新绑定到另一个节点，它要么被重启，要么终止，直到节点发生故障或被删除。

4.5.5 Pod的终止过程

Pod对象代表了在Kubernetes集群节点上运行的进程，它可能曾用于处理生产数据或向用户提供服务等，于是，当Pod本身不再具有存在的价值时，如何将其优雅地终止就显得尤为重要了，而用户也需要能够在正常提交删除操作后可以获知其何时开始终止并最终完成。操作中，当用户提交删除请求之后，系统就会进行强制删除操作的宽限期倒计时，并将TERM信息发送给Pod对象的每个容器中的主进程。宽限期倒计时结束后，这些进程将收到强制终止的KILL信号，Pod对象随即也将由API Server删除。如果在等待进程终止的过程中，kubelet或容器管理器发生了重启，那么终止操作会重新获得一个满额的删除宽限期并重新执行删除操作。

如图4-11所示，一个典型的Pod对象终止流程具体如下。

- 1) 用户发送删除Pod对象的命令。
- 2) API服务器中的Pod对象会随着时间的推移而更新，在宽限期内（默认为30秒），Pod被视为“dead”。
- 3) 将Pod标记为“Terminating”状态。
- 4) （与第3步同时运行）kubelet在监控到Pod对象转为“Terminating”状态的同时启动Pod关闭过程。
- 5) （与第3步同时运行）端点控制器监控到Pod对象的关闭行为时将其从所有匹配到此端点的Service资源的端点列表中移除。

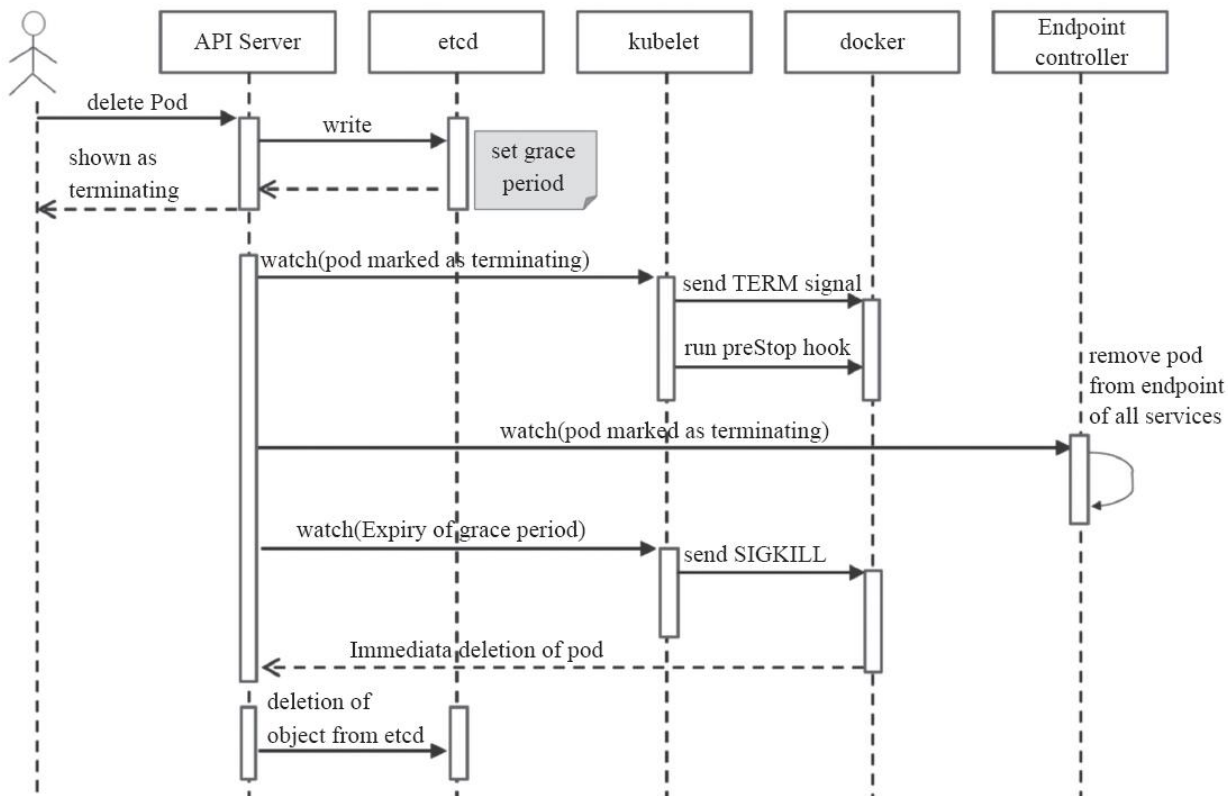


图4-11 Pod的终止过程

6) 如果当前Pod对象定义了preStop钩子处理器，则在其标记为“terminating”后即会以同步的方式启动执行；如若宽限期结束后，preStop仍未执行结束，则第2步会被重新执行并额外获取一个时长为2秒的小宽限期。

7) Pod对象中的容器进程收到TERM信号。

8) 宽限期结束后，若存在任何一个仍在运行的进程，那么Pod对象即会收到SIGKILL信号。

9) Kubelet请求API Server将此Pod资源的宽限期设置为0从而完成删除操作，它变得对用户不再可见。

默认情况下，所有删除操作的宽限期都是30秒，不过，`kubectl delete`命令可以使用“`--grace-period=<seconds>`”选项自定义其时长，若使用0值则表示直接强制删除指定的资源，不过，此时需要同时为命令使用“`--force`”选项。

4.6 Pod存活性探测

有不少应用程序长时间持续运行后会逐渐转为不可用状态，并且仅能通过重启操作恢复，Kubernetes的容器存活性探测机制可发现诸如此类的问题，并依据探测结果结合重启策略触发后续的行为。存活性探测是隶属于容器级别的配置，kubelet可基于它判定何时需要重启一个容器。

Pod spec为容器列表中的相应容器定义其专用的探针（存活性探测机制）即可启用存活性探测。目前，Kubernetes的容器支持存活性探测的方法包含以下三种：ExecAction、TCPSocketAction和HTTPGetAction。

4.6.1 设置exec探针

exec类型的探针通过在目标容器中执行由用户自定义的命令来判断容器的健康状态，若命令状态返回值为0则表示“成功”通过检测，其值均为“失败”状态。“**spec.containers.livenessProbe.exec**”字段用于定义此类检测，它只有一个可用属性“**command**”，用于指定要执行的命令。下面是定义在资源清单文件**liveness-exec.yaml**中的示例：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness-exec
  name: liveness-exec
spec:
  containers:
  - name: liveness-exec-demo
    image: busybox
    args: ["/bin/sh", "-c", "touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy;
      sleep 600"]
    livenessProbe:
      exec:
        command: ["test", "-e", "/tmp/healthy"]
```

上面的资源清单中定义了一个Pod对象，基于**busybox**镜像启动一个运行“**touch/tmp/healthy; sleep 60; rm-rf/tmp/healthy; sleep 600**”命令的容器，此命令在容器启动时创建**/tmp/healthy**文件，并于60秒之后将其删除。存活性探针运行“**test-e/tmp/healthy**”命令检查**/tmp/healthy**文件的存在性，若文件存在则返回状态码0，表示成功通过测试。

首先，执行类似如下的命令，创建Pod对象**liveness-exec**：

```
~]$ kubectl apply -f liveness-exec.yaml
pod "liveness-exec" created
```

在60秒之内使用“**kubectl describe pods/liveness-exec**”查看其详细信息，其存活性探测不会出现错误。而超过60秒之后，再次运行“**kubectl describe pods/liveness-exec**”查看其详细信息可以发现，存活性探测出现了故障，并且隔更长一段时间之后再查看甚至还可以看到容器重启的相关信息：

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
.....				
Warning	Unhealthy probe failed:	45s (x3 over 1m)	kubelet, node02.ilinux.io	Liveness
Normal	Pulling image "busybox"	14s (x2 over 2m)	kubelet, node02.ilinux.io	pulling
Normal	Killing container with id docker://liveness-demo:Container failed liveness probe.. Container will be killed and recreated.	14s	kubelet, node02.ilinux.io	Killing
Normal	Pulled Successfully pulled image "busybox"	10s (x2 over 2m)	kubelet, node02.ilinux.io	
.....				

另外，输出信息的“Containers”一段中还清晰地显示了容器健康状态检测及状态变化的相关信息：容器当前处于“**Running**”状态，但前一次是为“**Terminated**”，原因是退出码为137的错误信息，它表示进程是被外部信号所终止的。137事实上是由两部分数字之和生成的：128+**signum**，其中**signum**是导致进程终止的信号的数字标识，9表示SIGKILL，这意味着进程是被强行终止的：

liveness-exec-demo:	
.....	
State:	Running
.....	
Last State:	Terminated
Reason:	Error
Exit Code:	137
.....	
Ready:	True
Restart Count:	1
Liveness:	exec [test -e /tmp/healthy] delay=0s timeout=1s period=10s #success=1 #failure=3
.....	

待容器重启完成后再次查看，容器已经处于正常运行状态，直到文件再次被删除，存活性探测失败而重启。从下面的命令显示可以看出，**liveness-exec**在4分钟时间内已然重启了两次：

~]\$ kubectl get pods liveness-exec				
NAME	READY	STATUS	RESTARTS	AGE
liveness-exec	1/1	Running	2	4m

需要特别说明的是，**exec**指定的命令运行于容器中，会消耗容器的可用资源配额，另外，考虑到探测操作的效率本身等因素，探测操作的命令应该尽可能简单和轻量。

4.6.2 设置HTTP探针

基于HTTP的探测（HTTPGetAction）向目标容器发起一个HTTP请求，根据其响应码进行结果判定，响应码形如2xx或3xx时表示检测通过。“spec.containers.livenessProbe.httpGet”字段用于定义此类检测，它的可用配置字段包括如下几个。

·host<string>：请求的主机地址，默认为Pod IP；也可以在httpHeaders中使用“Host: ”来定义。

·port<string>：请求的端口，必选字段。

·httpHeaders<[]Object>：自定义的请求报文首部。

·path<string>：请求的HTTP资源路径，即URL path。

·scheme：建立连接使用的协议，仅可为HTTP或HTTPS，默认为HTTP。

下面是一个定义在资源清单文件liveness-http.yaml中的示例，它通过lifecycle中的postStart hook创建了一个专用于httpGet测试的页面文件healthz：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - name: liveness-http-demo
      image: nginx:1.12-alpine
      ports:
        - name: http
          containerPort: 80
      lifecycle:
        postStart:
          exec:
            command: ["/bin/sh", "-c", "echo Healthy > /usr/share/nginx/html/healthz"]
      livenessProbe:
        httpGet:
          path: /healthz
```

```
port: http
scheme: HTTP
```

上面清单文件中定义的httpGet测试中，请求的资源路径为“/healthz”，地址默认为Pod IP，端口使用了容器中定义的端口名称HTTP，这也是明确为容器指明要暴露的端口的用途之一。首先创建此Pod对象：

```
~]$ kubectl apply -f liveness-http.yaml
pod "liveness-http" created
```

而后查看其健康状态检测相关的信息，健康状态检测正常时，容器也将正常运行：

```
~]$ kubectl describe pods liveness-http
.....
Containers:
  liveness-http-demo:
    .....
    Port:          80/TCP
    State:          Running
      Started:      Thu, 21 Aug 2018 12:55:41 +0800
    Ready:          True
    Restart Count:  0
    Liveness:       http-get http://:http/healthz delay=0s timeout=1s period=10s
                   #success=1 #failure=3
    .....
```

接下来借助于“kubectl exec”命令删除经由postStart hook创建的测试页面healthz：

```
$ kubectl exec liveness-http rm /usr/share/nginx/html/healthz
```

而后再次使用“kubectl describe pods liveness-http”查看其详细的状态信息，事件输出中的信息可以表明探测测试失败，容器被杀掉后进行了重新创建：

```
Events:
.....
Warning Unhealthy          28s (x3 over 48s)  kubelet, node02.ilinux.io
Liveness
  probe failed: HTTP probe failed with statuscode: 404
```



```
Normal    Killing                28s          kubelet, node02.ilinux.io Killing
           container with id docker://liveness-demo:Container failed liveness probe..
Container will be killed and recreated.
```

一般来说，**HTTP**类型的探测操作应该针对专用的URL路径进行，例如，前面示例中特别为其准备的“/healthz”。另外，此URL路径对应的**Web**资源应该以轻量化的方式在内部对应用程序的各关键组件进行全面检测以确保它们可正常向客户端提供完整的服务。

需要注意的是，这种检测方式仅对分层架构中的当前一层有效，例如，它能检测应用程序工作正常与否的状态，但重启操作却无法解决其后端服务（如数据库或缓存服务）导致的故障。此时，容器可能会被一次次的重启，直到后端服务恢复正常为止。其他两种检测方式也存在类似的问题。

4.6.3 设置TCP探针

基于TCP的存活性探测（`TCPSocketAction`）用于向容器的特定端口发起TCP请求并尝试建立连接进行结果判定，连接建立成功即为通过检测。相比较来说，它比基于HTTP的探测要更高效、更节约资源，但精准度略低，毕竟连接建立成功未必意味着页面资源可用。“`spec.containers.livenessProbe.tcpSocket`”字段用于定义此类检测，它主要包含以下两个可用的属性。

- 1) `host<string>`: 请求连接的目标IP地址，默认为Pod IP。
- 2) `port<string>`: 请求连接的目标端口，必选字段。

下面是一个定义在资源清单文件`liveness-tcp.yaml`中的示例，它向Pod IP的80/tcp端口发起连接请求，并根据连接建立的状态判定测试结果：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
    - name: liveness-tcp-demo
      image: nginx:1.12-alpine
      ports:
        - name: http
          containerPort: 80
      livenessProbe:
        tcpSocket:
          port: http
```

这里不再给出其具体的创建与测试过程，有兴趣的读者可自行进行测试。

4.6.4 存活性探测行为属性

使用**kubectl describe**命令查看配置了存活性探测的Pod对象的详细信息时，其相关容器中会输出类似如下一行的内容：

```
Liveness:  exec [test -e /tmp/healthy] delay=0s timeout=1s period=10s  
           #success=1 #failure=3
```

它给出了探测方式及其额外的配置属性**delay**、**timeout**、**period**、**success**和**failure**及其各自的相关属性值。用户没有明确定义这些属性字段时，它们会使用各自的默认值，例如上面显示出的设定。这些属性信息可通过“**spec.containers.livenessProbe**”的如下属性字段来给出。

·**initialDelaySeconds<integer>**：存活性探测延迟时长，即容器启动多久之后再开始第一次探测操作，显示为**delay**属性；默认为0秒，即容器启动后立刻便开始进行探测。

·**timeoutSeconds<integer>**：存活性探测的超时时长，显示为**timeout**属性，默认为1s，最小值也为1s。

·**periodSeconds<integer>**：存活性探测的频度，显示为**period**属性，默认为10s，最小值为1s；过高的频率会对Pod对象带来较大的额外开销，而过低的频率又会使得对错误的反应不及时。

·**successThreshold<integer>**：处于失败状态时，探测操作至少连续多少次的成功才被认为是通过检测，显示为**#success**属性，默认值为1，最小值也为1。

·**failureThreshold**：处于成功状态时，探测操作至少连续多少次的失败才被视为是检测不通过，显示为**#failure**属性，默认值为3，最小值为1。

例如，这里可将4.6.1节中清单文件中定义的探测示例重新定义为如下所示的内容：

```
spec:
  containers:
  .....
    livenessProbe:
      exec:
        command: ["test", "-e", "/tmp/healthy"]
      initialDelaySeconds 5s
      timeoutSeconds 2s
      periodSeconds 5s
```

根据修改的清单再次创建Pod对象并进行效果测试，可以从输出的详细信息中看出**Liveness**已经更新到自定义的属性，其内容如下所示。具体过程这里不再给出，请感兴趣的读者自行测试。

```
Liveness: exec [test -e /tmp/healthy] delay=5s timeout=2s period=5s
           #success=1 #failure=3
```

4.7 Pod就绪性探测

Pod对象启动后，容器应用通常需要一段时间才能完成其初始化过程，例如加载配置或数据，甚至有些程序需要运行某类的预热过程，若在此阶段完成之前即接入客户端的请求，势必会因为等待太久而影响用户体验。因此，应该避免于Pod对象启动后立即让其处理客户端请求，而是等待容器初始化工作执行完成并转为“就绪”状态，尤其是存在其他提供相同服务的Pod对象的场景更是如此。

与存活性探测机制类似，就绪性探测是用来判断容器就绪与否的周期性（默认周期为10秒钟）操作，它用于探测容器是否已经初始化完成并可服务于客户端请求，探测操作返回“success”状态时，即为传递容器已经“就绪”的信号。

与存活性探测机制相同，就绪性探测也支持Exec、HTTP GET和TCP Socket三种探测方式，且各自的定义机制也都相同。但与存活性探测触发的操作不同的是，探测失败时，就绪性探测不会杀死或重启容器以保证其健康性，而是通知其尚未就绪，并触发依赖于其就绪状态的操作（例如，从Service对象中移除此Pod对象）以确保不会有客户端请求接入此Pod对象。不过，即便是在运行过程中，Pod就绪性探测依然有其价值所在，例如Pod A依赖到的Pod B因网络故障等原因而不可用时，Pod A上的服务应该转为未就绪状态，以免无法向客户端提供完整的响应。

将容器定义中的livenessProbe字段名替换为readinessProbe即可定义出就绪性探测的配置，一个简单的示例如下面的配置清单

（readiness-exec.yaml）所示，它会在Pod对象创建完成5秒钟后使用test-e/tmp/ready命令来探测容器的就绪性，命令执行成功即为就绪，探测周期为5秒钟：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness-exec
  name: readiness-exec
spec:
  containers:
  - name: readiness-demo
```

```
image: busybox
args: ["/bin/sh", "-c", "while true; do rm -f /tmp/ready; sleep 30; touch
/tmp/ready; sleep 300; done"]
readinessProbe:
  exec:
    command: ["test", "-e", "/tmp/ready"]
  initialDelaySeconds: 5
  periodSeconds: 5
```

首先，使用“`kubectl create`”命令将资源配置清单定义的资源创建到集群中：

```
~]$ kubectl create -f readiness-exec.yaml
pod/readiness-exec created
```

接着，运行“`kubectl get -w`”命令监视其资源变动信息，由如下命令结果可知，尽管Pod对象处于“**Running**”状态，但直到就绪探测命令执行成功后，Pod资源才转为“就绪”：

```
~]$ kubectl get pods -l test=readiness-exec -w
NAME          READY   STATUS    RESTARTS   AGE
readiness-exec 0/1     Running   0          15s
readiness-exec 1/1     Running   0          41s
```

另外，还可从Pod对象的详细信息中得到类似如下的表示其已经处于就绪状态的信息片断：

```
Ready:          True
Restart Count:  0
Readiness:      exec [test -e /tmp/ready] delay=5s timeout=1s period=5s
#success=1
#failure=3
```

这里需要特别提醒读者的是，未定义就绪性探测的Pod对象在Pod进入“**Running**”状态后将立即就绪，在容器需要时间进行初始化的场景中，在应用真正就绪之前必然无法正常响应客户端请求，因此，生产实践中，必须为关键性Pod资源中的容器定义就绪性探测机制。其探测机制的定义请参考4.6节中的定义。

4.8 资源需求及资源限制

在Kubernetes上，可由容器或Pod请求或消费的“计算资源”是指CPU和内存（RAM），这也是目前仅有的受支持的两种类型。相比较来说，CPU属于可压缩（compressible）型资源，即资源额度可按需收缩，而内存（当前）则是不可压缩型资源，对其执行收缩操作可能会导致某种程度的问题。

目前来说，资源隔离尚且属于容器级别，CPU和内存资源的配置需要在Pod中的容器上进行，每种资源均可由“requests”属性定义其请求的确保可用值，即容器运行可能用不到这些额度的资源，但用到时必须确保有如此多的资源可用，而“limits”属性则用于限制资源可用的最大值，即硬限制，如图4-12所示。不过，为了表述方便，人们通常仍然把资源配置称作Pod资源的请求和限制，只不过它是指Pod内所有容器上某种类型资源的请求和限制的总和。

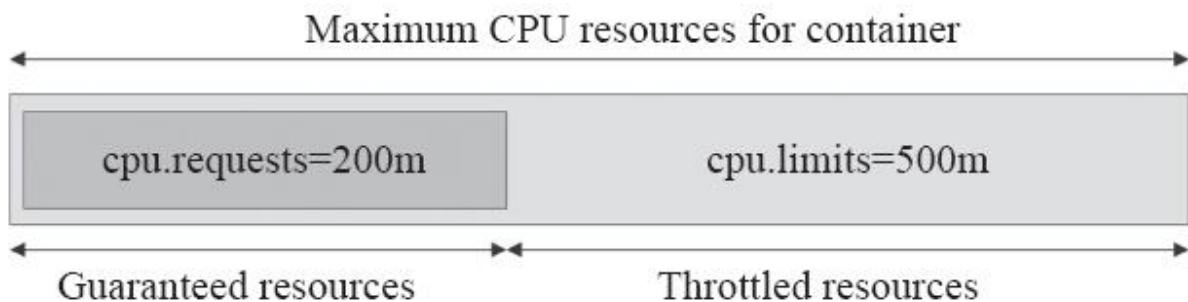


图4-12 容器资源需求及资源限制示意图

在Kubernetes系统上，1个单位的CPU相当于虚拟机上的1颗虚拟CPU（vCPU）或物理机上的一个超线程（Hyperthread，或称为一个逻辑CPU），它支持分数计量方式，一个核心（1core）相当于1000个微核心（millicores），因此500m相当于是0.5个核心，即二分之一个核心。内存的计量方式与日常使用方式相同，默认单位是字节，也可以使用E、P、T、G、M和K作为单位后缀，或Ei、Pi、Ti、Gi、Mi和Ki形式的单位后缀。

4.8.1 资源需求

下面的示例中，自主式Pod要求为stress容器确保128Mi的内存及五分之一CPU核心（200m）资源可用，它运行stress-ng镜像启动一个进程（-m 1）进行内存性能压力测试，满载测试时它也会尽可能多地占用CPU资源，另外再启动一个专用的CPU压力测试进程（-c 1）。stress-ng是一个多功能系统压力测试具，master/worker模型，Master为主进程，负责生成和控制子进程，worker是负责执行各类特定测试的子进程，例如测试CPU的子进程，以及测试RAM的子进程等：

```
apiVersion: v1
kind: Pod
metadata:
  name: stress-pod
spec:
  containers:
  - name: stress
    image: ikubernetes/ stress-ng
    command: ["/usr/bin/stress-ng", "-m 1", "-c 1", "-metrics-brief"]
    resources:
      requests:
        memory: "128Mi"
        cpu: "200m"
```

上面的配置清单中，其请求使用的CPU资源大小为200m，这意味着一个CPU核心足以确保其以期望的最快方式运行。另外，配置清单中期望使用的内存大小为128Mi，不过其运行时未必真的会用到这么多。考虑到内存为非压缩型资源，其超出指定的大小在运行时存在被OOM killer杀死的可能性，于是请求值也应该就是其理想中使用的内存空间上限。

接下来创建并运行此Pod对其资源限制效果进行检查。需要特别说明的是，笔者当前使用的系统环境中，每个节点的可用CPU核心数均为8，物理内存空间为16GB：

```
~]$ kubectl create -f pod-resources-test.yaml
```

而后在Pod资源的容器内运行top命令观察其CPU及内存资源的占用状态，如下所示，其中{stress-ng-vm}是执行内存压测的子进程，它

默认使用256m的内存空间，{stress-ng-cpu}是执行CPU压测的专用子进程：

```
~]$ kubectl exec stress-pod -- top
Mem: 2884676K used, 13531796K free, 27700K shrd, 2108K buff, 1701456K cached
CPU:  25% usr   0% sys   0% nic  74% idle   0% io   0% irq   0% sirq
Load average: 0.57 0.60 0.71 3/435 15
PID   PPID USER  STAT  VSZ   %VSZ CPU  %CPU COMMAND
9      8 root    R     262m   2%   6   13% {stress-ng-vm} /usr/bin/stress-ng
7      1 root    R    6888   0%   3   13% {stress-ng-cpu} /usr/bin/stress-ng
1      0 root    S    6244   0%   1    0% /usr/bin/stress-ng -c 1 -m 1 --met
.....
```

top命令的输出结果显示，每个测试进程的CPU占用率为13%（实际为12.5%），{stress-ng-vm}的内存占用量为262m（VSZ），此两项资源占用量都远超其请求的用量，原因是stress-ng会在可用的范围内尽量多地占用相关的资源。两个测试线程分布于两个CPU核心以满载的方式运行，系统共有8个核心，因此其使用率为25%（2/8）。另外，节点上的内存资源充裕，虽然容器的内存用量远超128M，但它依然可运行。一旦资源紧张时，节点仅保证容器有五分之一一个CPU核心可用，对于有着8个核心的节点来说，它的占用率为2.5%，于是每个进程为1.25%，多占用的资源会被压缩。内存为非可压缩型资源，所以此Pod在内存资源紧张时可能会因OOM被杀死（killed）。

对于压缩型的资源CPU来说，未定义其请求用量以确保其最小的可用资源时，它可能会被其他的Pod资源压缩至极低的水平，甚至会达到Pod不能够被调度运行的境地。而对于非压缩型资源来说，内存资源在任何原因导致的紧缺情形下都有可能相关的进程被杀死。因此，在Kubernetes系统上运行关键型业务相关的Pod时必须使用requests属性为容器定义资源的确保可用量。

集群中的每个节点都拥有定量的CPU和内存资源，调度Pod时，仅那些被请求资源的余量可容纳当前被调度的Pod的请求量的节点才可作为目标节点。也就是说，Kubernetes的调度器会根据容器的requests属性中定义的资源需求量来判定仅哪些节点可接收运行相关的Pod资源，而对于一个节点的资源来说，每运行一个Pod对象，其requests中定义的请求量都要被预留，直到被所有Pod对象瓜分完毕为止。

4.8.2 资源限制

容器的资源需求仅能达到为其保证可用的最少资源量的目的，它并不会限制容器的可用资源上限，因此对因应用程序自身存在Bug等多种原因而导致的系统资源被长时间占用的情况则无计可施，这就需要通过limits属性为容器定义资源的最大可用量。资源分配时，可压缩型资源CPU的控制阀可自由调节，容器进程无法获得超出其CPU配额的可用时间。不过，如果进程申请分配超出其limits属性定义的硬限制的内存资源时，它将被OOM killer杀死，不过，随后可能会被其控制进程所重启，例如，容器进程的Pod对象会被杀死并重启（重启策略为Always或OnFailure时），或者是容器进程的子进程被其父进程所重启。

下面的配置清单文件（memleak-pod.yaml）中定义了如何使用saadali/simmemleak镜像运行一个Pod对象，它模拟内存泄漏操作不断地申请使用内存资源，直到超出limits属性中memory字段设定的值而导致“OOMKilled”为止：

```
apiVersion: v1
kind: Pod
metadata:
  name: memleak-pod
  labels:
    app: memleak
spec:
  containers:
  - name: simmemleak
    image: saadali/simmemleak
    resources:
      requests:
        memory: "64Mi"
        cpu: "1"
      limits:
        memory: "64Mi"
        cpu: "1"
```

下面测试其运行效果，首先将配置清单中定义的资源复用下面的命令创建于集群中：

```
~]$ kubectl apply -f memleak-pod.yaml
pod/memleak created
```

Pod资源的默认重启策略为Always，于是在memleak因内存资源达到硬限制而被终止后会立即重启，因此用户很难观察到其因OOM而被杀死的相关信息。不过，多次重复地因为内存资源耗尽而重启会触发Kubernetes系统的重启延迟机制，即每次重启的时间间隔会不断地拉长。于是，用户看到的Pod资源的相关状态通常为“CrashLoopBackOff”：

```
~]$ kubectl get pods -l app=memleak
NAME          READY   STATUS             RESTARTS   AGE
memleak-pod   0/1     CrashLoopBackOff   1          24s
```

Pod资源首次的重启将在crash后立即完成，若随后再次crash，那么其重启操作会延迟10秒进行，随后的延迟时长会逐渐增加，依次为20秒、40秒、80秒、160秒和300秒，随后的延迟将固定在5分钟的时长之上而不再增加，直到其不再crash或者delete为止。describe命令可以显示其状态相关的详细信息，其部分内容如下所示：

```
~]$ kubectl describe pods memleak-pod
Name:          memleak-pod
.....
Last State:    Terminated
  Reason:      OOMKilled
  Exit Code:    137
  Started:     Wed, 02 May 2018 12:42:50 +0800
  Finished:    Wed, 02 May 2018 12:42:50 +0800
  Ready:       False
  Restart Count: 3
.....
```

如上述命令结果所显示的，OOMKilled表示容器因内存耗尽而被终止，因此，为limits属性中的memory设置一个合理值至关重要。与requests不同的是，limits并不会影响Pod的调度结果，也就是说，一个节点上的所有Pod对象的limits数量之和可以大于节点所拥有的资源量，即支持资源的过载使用（overcommitted）。不过，这么一来一旦资源耗尽，尤其是内存资源耗尽，则必然会有容器因OOMKilled而终止。

另外需要说明的是，Kubernetes仅会确保Pod能够获得它们请求（requests）的CPU时间额度，它们能否获得额外（throttled）的CPU

时间，则取决于其他正在运行的作业对CPU资源的占用情况。例如，对于总数为1000m的CPU资源来说，容器A请求使用200m，容器B请求使用500m，在不超出它们各自的最大限额的前提下，余下的300m在双方都需要时会以2：5（200m：500m）的方式进行配置。

4.8.3 容器的可见资源

细心的读者可能已经发现了这一点：于容器中运行`top`等命令观察资源可用量信息时，即便定义了`requests`和`limits`属性，虽然其可用资源受限于此两个属性中的定义，但容器中可见的资源量依然是节点级别的可用总量。例如，为前面定义的`stress-pod`添加如下`limits`属性定义：

```
limits:
  memory: "512Mi"
  cpu: "400m"
```

重新创建`stress-pod`对象，并于其容器内分别列出容器可见的内存和CPU资源总量，命令及结果如下所示：

```
~]$ kubectl exec stress-pod -- cat /proc/meminfo | grep ^MemTotal
MemTotal:          16416472 kB
$ kubectl exec stress-pod -- cat /proc/cpuinfo | grep -c ^processor
8
```

命令结果中显示其可用内存资源总量为16416472KB（16GB），CPU核心数为8个，这是节点级的资源数量，而非由容器的`limits`所定义的512Mi和400m。其实，这种结果不仅仅使得其查看命令的显示结果看起来有些奇怪，而且对有些容器应用的配置也会带来不小的负面影响。

较为典型的是在Pod中运行Java应用程序时，若未使用“-Xmx”选项指定JVM的堆内存可用总量，它默认会设置为主机内存总量的一个空间比例（如30%），这会导致容器中的应用程序申请内存资源时将会达到上限而转为OOMKilled。另外，即便使用了“-Xmx”选项设置其堆内存上限，但它对于非堆内存的可用空间不会产生任何限制作用，结果是仍然存在达到容器内存资源上限的可能性。

另一个颇具代表性的场景是于Pod中运行的nginx应用，在配置参数`worker_processes`的值为“auto”时，主进程会创建与Pod中能够访问到的CPU核心数相同数量的worker进程。若Pod的实际可用CPU核心远低

于主机级别的数量时，那么这种设置在较大的并发访问负荷下会导致严重的资源竞争，并将带来更多的内存资源消耗。一个较为妥当的解决方案是使用**Downward API**将**limits**定义的资源量暴露给容器，这一点将在后面的章节中予以介绍。

4.8.4 Pod的服务质量类别

前面曾提到过，Kubernetes允许节点资源对limits的过载使用，这意味着节点无法同时满足其上的所有Pod对象以资源满载的方式运行。于是，在内存资源紧缺时，应该以何种次序先后终止哪些Pod对象？Kubernetes无法自行对此做出决策，它需要借助于Pod对象的优先级完成判定。根据Pod对象的requests和limits属性，Kubernetes将Pod对象归类到BestEffort、Burstable和Guaranteed三个服务质量（Quality of Service, QoS）类别下，具体说明如下。

- Guaranteed**: 每个容器都为CPU资源设置了具有相同值的requests和limits属性，以及每个容器都为内存资源设置了具有相同值的requests和limits属性的Pod资源会自动归属于此类别，这类Pod资源具有最高优先级。

- Burstable**: 至少有一个容器设置了CPU或内存资源的requests属性，但不满足Guaranteed类别要求的Pod资源将自动归属于此类别，它们具有中等优先级。

- BestEffort**: 未为任何一个容器设置requests或limits属性的Pod资源将自动归属于此类别，它们的优先级为最低级别。

内存资源紧缺时，BestEffort类别的容器将首当其冲地被终止，因为系统不为其提供任何级别的资源保证，但换来的好处是，它们能够在可用时做到尽可能多地占用资源。若已然不存任何BestEffort类别的容器，则接下来是有着中等优先级的Burstable类别的Pod被终止。Guaranteed类别的容器拥有最高优先级，它们不会被杀死，除非其内存资源需求超限，或者OOM时没有其他更低优先级的Pod资源存在。

每个运行状态容器都有其OOM得分，得分越高越会被优先杀死。OOM得分主要根据两个纬度进行计算：由QoS类别继承而来的默认分值和容器的可用内存资源比例。同等类别的Pod资源的默认分值相同，下面的代码片段取自pkg/kubelet/qos/policy.go源码文件，它们定义的是各种类别的Pod资源的OOM调节（Adjust）分值，即默认分值。其中，Guaranteed类别的Pod资源的Adjust分值为-998，而BestEffort类

别的默认分值为1000，Burstable类别的Pod资源的Adjust分值则经由相应的算法计算得出：

```
const (  
    PodInfraOOMAdj      int = -998  
    KubeletOOMScoreAdj  int = -999  
    DockerOOMScoreAdj   int = -999  
    KubeProxyOOMScoreAdj int = -999  
    guaranteedOOMScoreAdj int = -998  
    besteffortOOMScoreAdj int = 1000  
)
```

因此，同等级别优先级的Pod资源在OOM时，与自身的requests属性相比，其内存占用比例最大的Pod对象将被首先杀死。例如，图4-13中的同属于Burstable类别的Pod A将先于Pod B被杀死，虽然其内存用量小，但与自身的requests值相比，它的占用比例95%要大于Pod B的80%。

需要特别说明的是，OOM是内存耗尽时的处理机制，它们与可压缩型资源CPU无关，因此CPU资源的需求无法得到保证时，Pod仅仅是暂时获取不到相应的资源而已。

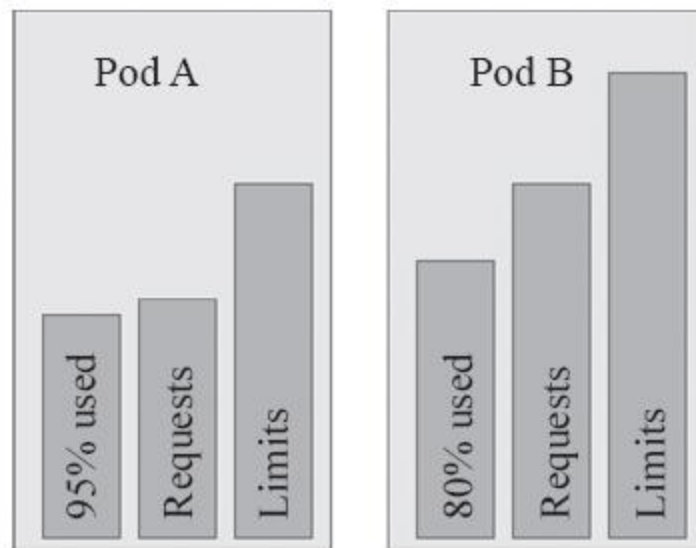


图4-13 资源需求、资源限额及OOM

4.9 本章小结

本章介绍了Pod资源的基础概念、分布式系统的设计模式、Pod的基础管理操作、如何定义和管理容器、资源标签和标签选择器、资源注解等，详细讲解了Pod生命周期中的事件、容器的存活性探测和就绪性探测机制等话题。

- Pod就是联系紧密的一组容器，它们共享Network、UTS和IPC名称空间及存储卷资源。

- 分布式系统设计主要有Sidecar、Ambassador和Adapter三种主要模式。

- Kubernetes资源对象的管理操作基本上是由增、删、改和查等操作组成的，并且支持陈述式命令、陈述式对象配置和声明式对象配置三种管理方式。

- Pod的核心目标在于运行容器，容器的定制配置常见的包括暴露端口及传递环境变量等。

- 标签是附加在Kubernetes系统上的键值类型的元数据，而标签选择器是基本等值或集合关系的标签过滤机制；注解类似于标签，但不能被用于标签选择器。

- Pod的生命周期中可能存在多种类型的操作，但运行主容器是其核心任务。

- 存活性探测及就绪性探测是辅助判定容器状态的重要工具。

- 资源需求及资源限制是管理Pod对象系统资源分配的有效方式。

第5章 Pod控制器

自主式Pod对象由调度器绑定至目标工作节点后即由相应节点上的kubernetes负责监控其容器的存活性，容器主进程崩溃后，kubernetes能够自动重启相应的容器。不过，kubernetes对非主进程崩溃类的容器错误却无从感知，这依赖于用户为Pod资源对象自定义的存活性探测

(liveness probe) 机制，以便kubernetes能够探知到此类故障。然而，在Pod对象遭到意外删除，或者工作节点自身发生故障时，又该如何处理呢？

kubernetes是Kubernetes集群节点代理程序，它在每个工作节点上都运行着一个实例。因而，集群中的某工作节点发生故障时，其kubernetes也必将不再可用，于是，节点上的Pod资源的健康状态将无从得到保证，也无法再由kubernetes重启。此种场景中的Pod存活性一般要由工作节点之外的Pod控制器来保证。事实上，遭到意外删除的Pod资源的恢复也依赖于其控制器。

Pod控制器由master的kube-controller-manager组件提供，常见的此类控制器有ReplicationController、ReplicaSet、Deployment、DaemonSet、StatefulSet、Job和CronJob等，它们分别以不同的方式管理Pod资源对象。实践中，对Pod对象的管理通常都是由某种控制器的特定对象来实现的，包括其创建、删除及重新调度等操作。本章将逐一讲解常用的Pod控制器资源。

5.1 关于Pod控制器

我们可以把API Server类比成一个存储对象的数据库系统，它向客户端提供了API，并负责存储由用户创建的各种资源对象，至于各对象的当前状态如何才能符合用户期望的状态，则需要交由另一类称为控制器的组件来负责完成。Kubernetes提供了众多的控制器来管理各种类型的资源，如Node Lifecycle Controller、Namespace Controller、Service Controller和Deployment Controller等，它们的功用几乎可以做到见名知义。创建完成后，每一个控制器对象都可以通过内部的和解循环（reconciliation loop），不间断地监控着由其负责的所有资源并确保其处于或不断地逼近用户定义的目标状态。

尽管能够由kubelet为其提供自愈能力，但在节点宕机时，自主式Pod对象的重建式自愈机制则需要由Pod控制器对象负责提供，并且由它来负责实现生命周期中的各类自动管理行为，如创建及删除等。

5.1.1 Pod控制器概述

Master的各组件中，API Server仅负责将资源存储于etcd中，并将其变动通知给各相关的客户端程序，如kubelet、kube-scheduler、kube-proxy和kube-controller-manager等，kube-scheduler监控到处于未绑定状态的Pod对象出现时遂启动调度器为其挑选适配的工作节点，然而，Kubernetes的核心功能之一还在于要确保各资源对象的当前状态

（status）以匹配用户期望的状态（spec），使当前状态不断地向期望状态“和解”（reconciliation）来完成容器应用管理，而这些则是kube-controller-manager的任务。kube-controller-manager是一个独立的单体守护进程，然而它包含了众多功能不同的控制器类型分别用于各类和解任务，如图5-1所示。

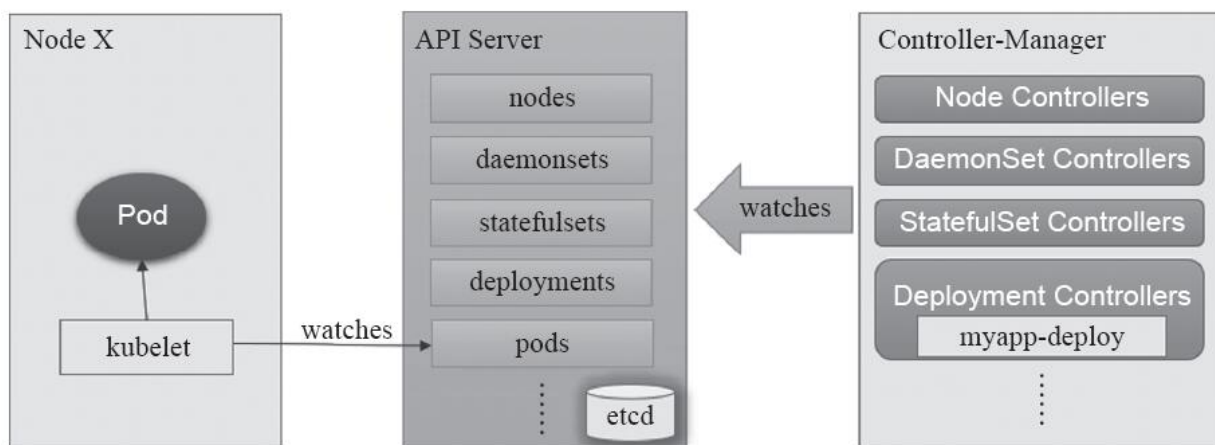


图5-1 kube-controller-manager及其控制器



提示 Kubernetes可用的控制器有attachdetach、bootstrapsigner、clusterrole-aggregation、cronjob、csrapproving、csrcleaner、csrsigning、daemonset、deployment、disruption、endpoint、garbagecollector、horizontalpodautoscaling、job、namespace、node、persistentvolume-binder、persistentvolume-expander、podgc、pvc-protection、replicaset、replication-controller、resourcequota、route、service、serviceaccount、serviceaccount-token、statefulset、tokencleaner和ttl等数十种。

创建为具体的控制器对象之后，每个控制器均通过API Server提供的接口持续监控相关资源对象的当前状态，并在因故障、更新或其他原因导致系统状态发生变化时，尝试让资源的当前状态向期望状态迁移和逼近。简单来说，每个控制器对象运行一个和解循环负责状态和解，并将目标资源对象的当前状态写入到其status字段中。控制器的“和解”循环如图5-2所示。

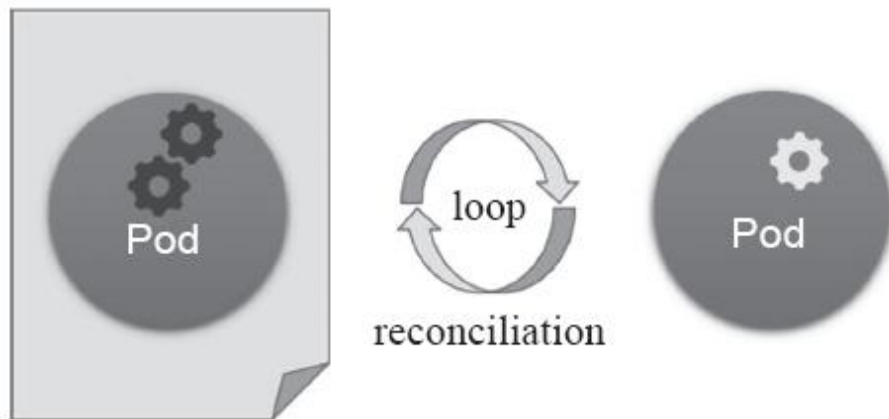


图5-2 控制器的“和解”循环


List-Watch是Kubernetes实现的核心机制之一，在资源对象的状态发生变动时，由API Server负责写入etcd并通过水平触发（level-triggered）机制主动通知给相关的客户端程序以确保其不会错过任何一个事件。控制器通过API Server的watch接口实时监控目标资源对象的变动并执行和解操作，但并不会与其他控制器进行任何交互，甚至彼此之间根本就意识不到对方的存在。

工作负载（workload）一类的控制器资源类型包括ReplicationController、ReplicaSet、Deployment、DaemonSet、StatefulSet、Job和CronJob等，它们分别代表了一种类型的Pod控制器资源，各类型的功用在3.1.1节中已经给出过说明。本章后面的篇幅主要介绍各控制器的特性及其应用，不过StatefulSet控制器依赖于存储卷资源，因此它将单独在存储卷之后的章节中给予介绍。

5.1.2 控制器与Pod对象

Pod控制器资源通过持续性地监控集群中运行着的Pod资源对象来确保受其管控的资源严格符合用户期望的状态，例如资源副本的数量要精确符合期望等。通常，一个Pod控制器资源至少应该包含三个基本的组成部分。

- 标签选择器：匹配并关联Pod资源对象，并据此完成受其管控的Pod资源计数。
- 期望的副本数：期望在集群中精确运行着的Pod资源的对象数量。
- Pod模板：用于新建Pod资源对象的Pod模板资源。

 **注意** DaemonSet用于确保集群中的每个工作节点或符合条件的每个节点上都运行着一个Pod副本，而不是某个精确的数量值，因此不具有上面组成部分中的第二项。

例如，一个如图5-3所示的Deployment控制器资源使用的标签选择器为“role=be-eshop”，它期望相关的Pod资源副本数量精确为3个，少于此数量的缺失部分将由控制器通过Pod模板予以创建，而多出的副本也将由控制器负责终止及删除。

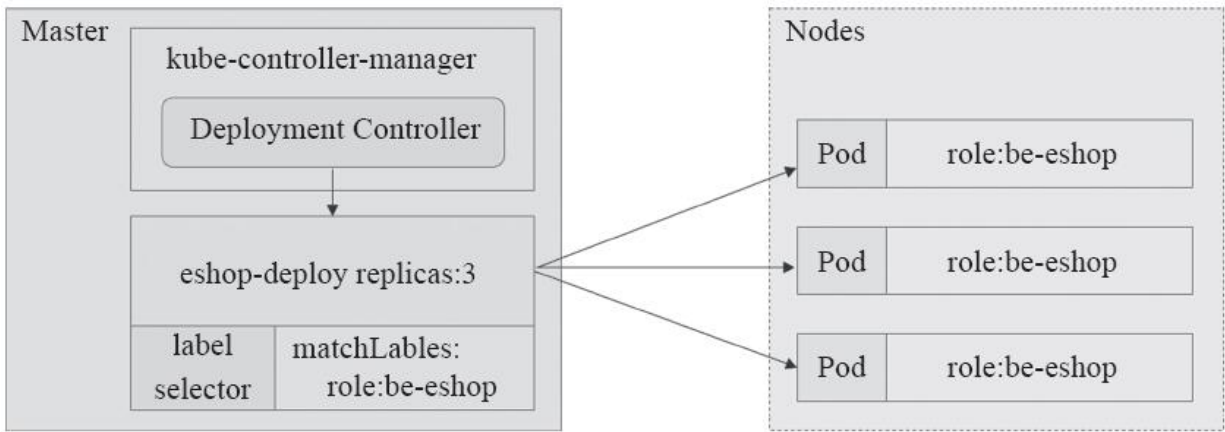


图5-3 Deployment控制器示例

5.1.3 Pod模板资源

PodTemplate是Kubernetes API的常用资源类型，常用于为控制器指定自动创建**Pod**资源对象时所需的配置信息。因为要内嵌于控制器中使用，所以**Pod**模板的配置信息中不需要**apiVersion**和**kind**字段，但除此之外的其他内容与定义自主式**Pod**对象所支持的字段几乎完全相同，这包括**metadata**和**spec**及其内嵌的其他各个字段。**Pod**控制器类资源的**spec**字段通常都要内嵌**replicas**、**selector**和**template**字段，其中**template**即为**Pod**模板的定义。下面是一个定义在**ReplicaSet**资源中的模板资源示例：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rs-demo
  template:
    metadata:
      labels:
        app: rs-demo
    spec:
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
          ports:
            - name: http
              containerPort: 80
```

如上示例中，**spec.template**字段在定义时仅给出了**metadata**和**spec**两个字段，它的使用方法与自主式**Pod**资源完全相同。后面讲到控制器的章节时会反复用到**Pod**模板资源。

5.2 ReplicaSet控制器

Kubernetes较早期的版本中仅有ReplicationController一种类型的Pod控制器，后来的版本中陆续引入了更多的控制器实现，这其中就包括用来取代ReplicationController的新一代实现ReplicaSet。事实上，除了额外支持基于集合（set-based）的标签选择器，以及它的滚动更新（Rolling-Update）机制要基于更高级的控制Deployment实现之外，目前的ReplicaSet的其余功能基本上与ReplicationController相同。考虑到Kubernetes强烈推荐使用ReplicaSet控制器，且表示ReplicationController不久后即将废弃，这里就重点介绍ReplicaSet控制器。

5.2.1 ReplicaSet概述

ReplicaSet（简称RS）是Pod控制器类型的一种实现，用于确保由其管控的Pod对象副本数在任一时刻都能精确满足期望的数量。如图5-4所示，ReplicaSet控制器资源启动后会查找集群中匹配其标签选择器的Pod资源对象，当前活动对象的数量与期望的数量不吻合时，多则删除，少则通过Pod模板创建以补足，等Pod资源副本数量符合期望值后即进入下一轮和解循环。

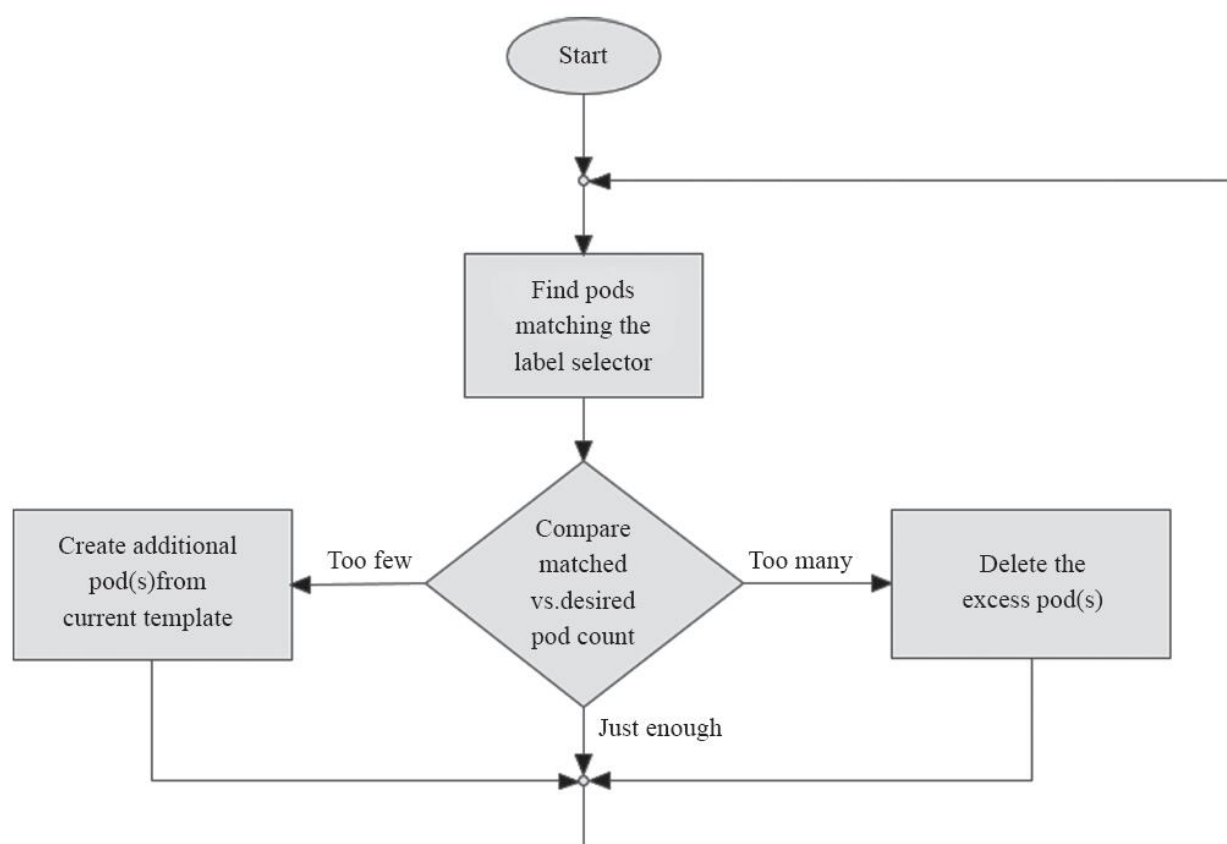


图5-4 ReplicaSet的控制循环（图片来源：《Kubernetes in action》）

ReplicaSet的副本数量、标签选择器甚至是Pod模板都可以随时按需进行修改，不过仅改动期望的副本数量会对现存的Pod副本产生直接影响。修改标签选择器可能会使得现有的Pod副本的标签变得不再匹配，此时ReplicaSet控制器要做的不过是不再计入它们而已。另外，在创建完成后，ReplicaSet也不会再关注Pod对象中的实际内容，因此Pod模板的改动也只会对后来新建的Pod副本产生影响。

相比较于手动创建和管理Pod资源来说，ReplicaSet能够实现以下功能。

- 确保Pod资源对象的数量精确反映期望值：ReplicaSet需要确保由其控制运行的Pod副本数量精确吻合配置中定义的期望值，否则就会自动补足所缺或终止所余。

- 确保Pod健康运行：探测到由其管控的Pod对象因其所在的工作节点故障而不可用时，自动请求由调度器于其他工作节点创建缺失的Pod副本。

- 弹性伸缩：业务规模因各种原因时常存在明显波动，在波峰或波谷期间，可以通过ReplicaSet控制器动态调整相关Pod资源对象的数量。此外，在必要时还可以通过HPA（HorizontalPodAutoscaler）控制器实现Pod资源规模的自动伸缩。

5.2.2 创建ReplicaSet

类似于Pod资源，创建ReplicaSet控制器对象同样可以使用YAML或JSON格式的清单文件定义其配置，而后使用相关的创建命令来完成资源创建。前面5.1.3节中给出的示例清单就是一个简单的ReplicaSet的定义。它也由kind、apiVersion、metadata、spec和status这5个一级字段组成，其中status为只读字段，因此需要在清单文件中配置的仅为前4个字段。它的spec字段一般嵌套使用以下几个属性字段。

·replicas<integer>: 期望的Pod对象副本数。

·selector<Object>: 当前控制器匹配Pod对象副本的标签选择器，支持matchLabels和matchExpressions两种匹配机制。

·template<Object>: 用于补足Pod副本数量时使用的Pod模板资源。

·minReadySecond<integer>s: 新建的Pod对象，在启动后的多长时间内如果其容器未发生崩溃等异常情况即被视为“就绪”；默认为0秒，表示一旦就绪性探测成功，即被视作可用。

将5.1.3节中的示例保存于资源清单文件中，例如rs-example.yaml，而后即可使用如下命令将其创建：

```
~]$ kubectl apply -f rs-example.yaml
replicaset.apps "rs-example" created
```

集群中当前没有标签为“app: rs-demo”的Pod资源存在，因此rs-example需要按照replicas字段的定义创建它们，名称以其所属的控制器名称为前缀。这两个Pod资源目前都处于ContainerCreating状态，即处于容器创建过程中，待创建过程完成后，其状态即转为Running，Pod也将转变为“READY”：

```
~]# kubectl get pods -l app=rs-demo
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

rs-example-p66nv	0/1	ContainerCreating	0	8s
rs-example-rdm7q	0/1	ContainerCreating	0	8s

接下来可以使用“`kubectl get replicaset`”命令查看ReplicaSet控制器资源的相关状态。下面的命令结果显示它已经根据清单中配置的Pod模板创建了2个Pod资源，不过这时它们尚未创建完成，因此仍为“READY”：

```
~]$ kubectl get replicaset rs-example -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES	SELECTOR
rs-example	2	2	0	2m	nginx	ikubernetes/myapp:v1	app=rs-demo

经由控制器创建与用户自主创建的Pod对象的功能并无二致，但其自动和解的功能在很大程度上能为用户省去不少的管理精力，这也是使得Kubernetes系统之上的应用程序变得拥有自愈能力的主要保障。

5.2.3 ReplicaSet管控下的Pod对象

5.2.2节中创建的rc-example通过标签选择器将拥有“app=rs-demo”标签的Pod资源收归于麾下，并确保其数量精确符合所期望的数目，使用标签选择器显示出的Pod资源列表也能验证这一点。然而，实际中存在着不少可能导致Pod对象数目与期望值不符合的可能性，如Pod对象的意外删除、Pod对象标签的变动（已有的Pod资源变得不匹配控制器的标签选择器，或者外部的Pod资源标签变得匹配到了控制器的标签选择器）、控制器的标签选择器变动，甚至是工作节点故障等。ReplicaSet控制器的和解循环过程能够实时监控到这类异常，并及时启动和解操作。

1. 缺少Pod副本

任何原因导致的相关Pod对象丢失，都会由ReplicaSet控制器自动补足。例如，手动删除上面列出的一个Pod对象，命令如下：

```
~]$ kubectl delete pods rs-example-rdm7q
pod "rs-example-rdm7q" deleted
```

再次列出相关Pod对象的信息，可以看到被删除的rs-example-rdm7q进入了终止过程，而新的Pod对象rs-example-l4gkp正在被rs-example控制器创建：

```
~]$ kubectl get pods -l app=rs-demo -o wide
```

NAME	READY	STATUS	RESTARTS	AGE
rs-example-l4gkp	0/1	ContainerCreating	0	1s
rs-example-p66nv	1/1	Running	0	39m
rs-example-rdm7q	0/1	Terminating	0	39m

另外，强行修改隶属于控制器rs-example的某Pod资源（匹配于标签控制器）的标签，会导致它不再被控制器作为副本计数，这也将触发控制器的Pod对象副本缺失补足机制。例如，将rs-example-p66nv的标签app的值置空：

```
~]$ kubectl label pods rs-example-p66nv app= --overwrite
pod "rs-example-p66nv" labeled
```

列出rs-example相关Pod对象的信息，发现rs-example-p66nv已经消失不见，并且正在创建新的对象副本。

```
~]$ kubectl get pods -l app=rs-demo
NAME                READY   STATUS             RESTARTS   AGE
rs-example-4bqzv    0/1     ContainerCreating   0           2s
rs-example-l4gkp    1/1     Running             0           5m
```

由此可见，修改Pod资源的标签即可将其从控制器的管控之下移出，当然，修改后的标签如果又能被其他控制器资源的标签选择器所命中，则此时它又成了隶属于另一控制器的副本。如果修改其标签后的Pod对象不再隶属于任何控制器，那么它将成为自主式Pod，与此前手动直接创建的Pod对象的特性相同，即误删除或所在的工作节点故障都会造成其永久性的消失。

2.多出Pod副本

一旦被标签选择器匹配到的Pod资源数量因任何原因超出期望值，多余的部分都将被控制器自动删除。例如，为pod-example手动为其添加“app: rs-demo”标签：

```
~]$ kubectl label pods pod-example app=rs-demo
pod "pod-example" labeled
```

再次列出相关的Pod资源，可以看到rs-example控制器启动了删除多余Pod的操作，pod-example正处于终止过程中：

```
~]$ kubectl get pods -l app=rs-demo
NAME                READY   STATUS             RESTARTS   AGE
pod-example         1/1     Terminating       0           2m
rs-example-4bqzv    1/1     Running             0           17m
rs-example-l4gkp    1/1     Running             0           22m
```

这就意味着，任何自主式的或本隶属于其他控制器的Pod资源其标签变动的结果一旦匹配到了其他的副本数足额的控制器，就会导致

这类Pod资源被删除。

3.查看Pod资源变动的相关事件

“`kubectl describe replicaset`”命令可打印出ReplicaSet控制器的详细状态，从下面命令结果中Events一段也可以看出，rs-example执行了Pod资源的创建和删除操作，为的就是确保其数量的精确性。

```
~]$ kubectl describe replicaset/rs-example
Name:          rs-example
Namespace:     default
Selector:      app=rs-demo
Labels:        app=rs-demo
.....
Events:
  Type      Reason             Age   From                      Message
  ----      -
  Normal    SuccessfulCreate   21m   replicaset-controller     Created pod: rs-example-l4gkp
  Normal    SuccessfulDelete   4m    replicaset-controller     Deleted pod: pod-example
  Normal    SuccessfulCreate   26m   replicaset-controller     Created pod: rs-example-4bqzv
```

事实上，ReplicaSet控制器能对Pod对象数目的异常及时做出响应，是因为它向API Server注册监听（watch）了相关资源及其列表的变动信息，于是API Server会在变动发生时立即通知给相关的监听客户端。

而因节点自身故障而导致的Pod对象丢失，ReplicaSet控制器一样会使用补足资源的方式进行处理，这里不再详细说明其过程。有兴趣的读者可通过直接关掉类似上面Pod对象运行所在的某一个节点来检验其处理过程。

5.2.4 更新ReplicaSet控制器

ReplicaSet控制器的核心组成部分是标签选择器、副本数量及Pod模板，但更新操作一般是围绕replicas和template两个字段值进行的，毕竟改变标签选择器的需求几乎不存在。改动Pod模板的定义对已经创建完成的活动对象无效，但在用户逐个手动关闭其旧版本的Pod资源后就能以新代旧，实现控制器下应用版本的滚动升级。另外，修改副本的数量也就意味着应用规模的扩展（提升期望的副本数量）或收缩（降低期望的副本数量）。这两种操作也是系统运维人员日常维护工作的重要组成部分。

1.更改Pod模板：升级应用

ReplicaSet控制器的Pod模板可随时按需修改，但它仅影响这之后由其新建的Pod对象，对已有的副本不会产生作用。大多数情况下，用户需要改变的通常是模板中的容器镜像文件及其相关的配置以实现应用的版本升级。下面的示例清单文件片断（rs-example-v2.yaml）中的内容与之前版本（rs-example.yaml）的唯一不同之处也仅在于镜像文件的改动：

```
containers:
- name: nginx
  image: ikubernetes/myapp:v2
  ports:
  - name: http
    containerPort: 80
```

对新版本的清单文件执行“kubectl apply”或“kubectl replace”命令即可完成rs-example控制器的修改操作：

```
~]$ kubectl replace -f rs-example-v2.yaml
replicaset.apps "rs-example" replaced
```

不过，控制器rs-example管控的现存Pod对象使用的仍然是原来版本中定义的镜像：

```
~]$ kubectl get pods -l app=rs-demo -o \
    custom-columns=Name:metadata.name,Image:spec.containers[0].image
Name          Image
rs-example-4bqzv  ikubernetes/myapp:v1
rs-example-l4gkp  ikubernetes/myapp:v1
```

此时，手动删除控制器现有的Pod对象（或修改与其匹配的控制器标签选择器的标签），并由控制器基于新的Pod模板自动创建出足额的Pod副本，即可完成一次应用的升级。新旧更替的过程支持如下两类操作方式。

·一次性删除控制器相关的所有Pod副本或更改相关的标签：剧烈更替，可能会导致Pod中的应用短时间不可访问（如图5-5所示）；生产实践中，此种做法不可取。

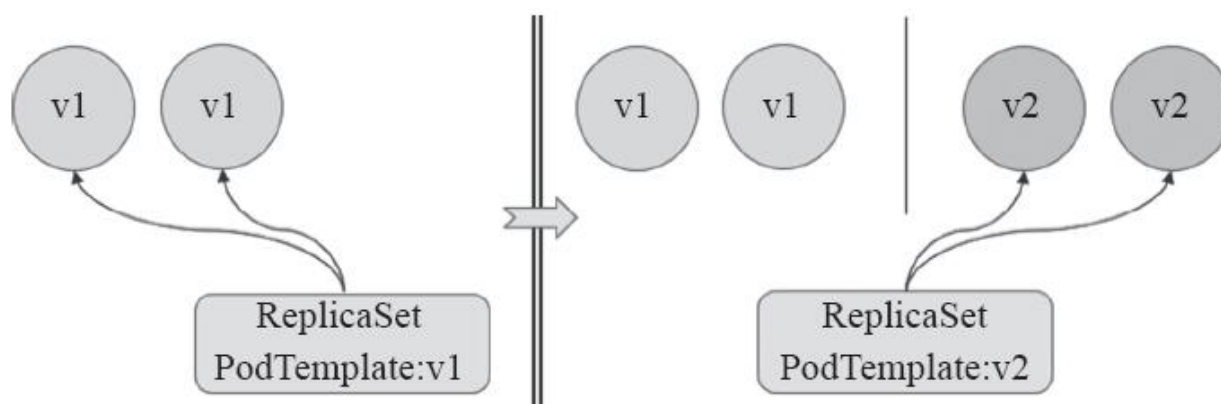


图5-5 直接更替所有Pod资源

·分批次删除旧有的Pod副本或更改其标签（待控制器补足后再删除另一批）：滚动更替，更替期间新旧版本共存（如图5-6所示）。

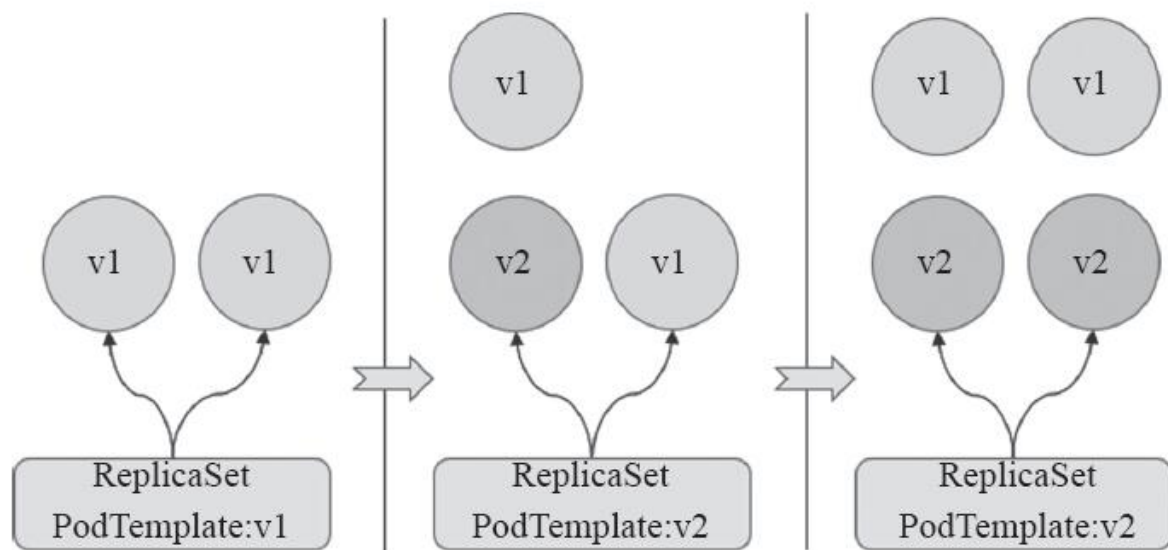


图5-6 滚动更替

例如，这里采用第一种方式进行操作，一次性删除rs-example相关的所有Pod副本：

```
~]$ kubectl delete pods -l app=rs-demo
pod "rs-example-4bqzv" deleted
pod "rs-example-l4gkp" deleted
```

再次列出rs-example控制器相关的Pod及其容器镜像版本时可以发现，使用新版本镜像的Pod已经创建完成：

```
~]$ kubectl get pods -l app=rs-demo -o \
custom-columns=Name:metadata.name,Image:spec.containers[0].image
Name          Image
rs-example-4bwqq  ikubernetes/myapp:v2
rs-example-hqgnh  ikubernetes/myapp:v2
```

必要时，用户还可以将Pod模板改回旧的版本进行应用的“降级”或“回滚”，它的操作过程与上述过程基本类似。事实上，修改Pod模板时，不仅仅能替换镜像文件的版本，甚至还可以将其替换成其他正在运行着的、完全不同应用的镜像，只不过此类需求并不多见。若同时改动的还有Pod模板中的其他字段，那么在新旧更替的过程中，它们也将随之被应用。

以上操作只为说明应用部署的方式，实际使用时还需要更为完善的机制。即便是仅执行了一到多次删除操作，手动执行更替操作也并非一项轻松的任务，幸运的是，更高级别的Pod控制器Deployment能够自动实现更完善的滚动更新和回滚，并为用户提供自定义更新策略的接口。而且，经过精心组织的更新操作还可以实现诸如蓝绿部署（Blue/Green Deployment）、金丝雀部署（Canary Deployment）和灰度部署等，这些内容将在后面章节中详细展开说明。

2. 扩容和缩容

改动ReplicaSet控制器对象配置中期望的Pod副本数量（replicas字段）会由控制器实时做出响应，从而实现应用规模的水平伸缩。replicas的修改及应用方式同Pod模板，不过，kubectl还提供了—个专用的子命令scale用于实现应用规模的伸缩，它支持从资源清单文件中获取新的目标副本数量，也可以直接在命令行通过“--replicas”选项进行读取，例如将rs-example控制器的Pod副本数量提升至5个：

```
~]$ kubectl scale replicaset rs-example --replicas=5
replicaset.extensions "rs-example" scaled
```

由下面显示的rs-example资源的状态可以看出，将其Pod资源副本数量扩展至5个的操作已经成功完成：

```
~]$ kubectl get replicaset rs-example
NAME      DESIRED  CURRENT  READY  AGE
rs-example 5         5        5      12h
```

收缩规模的方式与扩展相同，只需要明确指定目标副本数量即可。例如：

```
~]$ kubectl scale replicaset rs-example --replicas=3
replicaset.extensions "rs-example" scaled
```

另外，kubectl scale命令还支持在现有Pod副本数量符合指定的值时才执行扩展操作，这仅需要为命令使用“--current-replicas”选项即可。例如，下面的命令表示如果rs-example目前的Pod副本数量为2，就将其扩展至4个：

```
~]$ kubectl scale replicaset rs-example --current-replicas=2 --replicas=4  
error: Expected replicas to be 2, was 3
```

但由于rs-example控制器现存的副本数量是3个，因此上面的扩展操作未执行并返回了错误提示。



注意 如果ReplicaSet控制器管控的是有状态的应用，例如主从架构的Redis集群，那么上述这些升级、降级、扩展和收缩的操作都需要精心编排和参与才能进行，不过，这也在一定程度上降低了Kubernetes容器编排的价值和意义。好在，它提供了StatefulSet资源来应对这种需求，因此，ReplicaSet通常仅用于管理无状态的应用，如HTTP服务程序等。

5.2.5 删除ReplicaSet控制器资源

使用`kubectl delete`命令删除ReplicaSet对象时默认会一并删除其管控的各Pod对象。有时，考虑到这些Pod资源未必由其创建，或者即便由其创建却也并非其自身的组成部分，故而可以为命令使用“`--cascade=false`”选项，取消级联，删除相关的Pod对象，这在Pod资源后续可能会再次用到时尤为有用。例如，删除rs控制器rs-example：

```
~]$ kubectl delete replicaset rs-example --cascade=false  
replicaset.extensions "rs-example" deleted
```

删除操作完成后，此前由rs-example控制器管控的各Pod对象仍处于活动状态，但它们变成了自主式Pod资源，用户需要自行组织和维护好它们。



提示 后续讲到的各Pod控制器的删除方式都与ReplicaSet类似，这里就不再分别进行说明了。

尽管ReplicaSet控制器功能强大，但在实践中，它却并非为用户直接使用的控制器，而是要由比其更高级抽象的Deployment控制器对象来调用。

5.3 Deployment控制器

Deployment（简写为deploy）是Kubernetes控制器的又一种实现，它构建于ReplicaSet控制器之上，可为Pod和ReplicaSet资源提供声明式更新。相比较而言，Pod和ReplicaSet是较低级别的资源，它们很少被直接使用。Deployment、ReplicaSet和Pod的关系如图5-7所示。

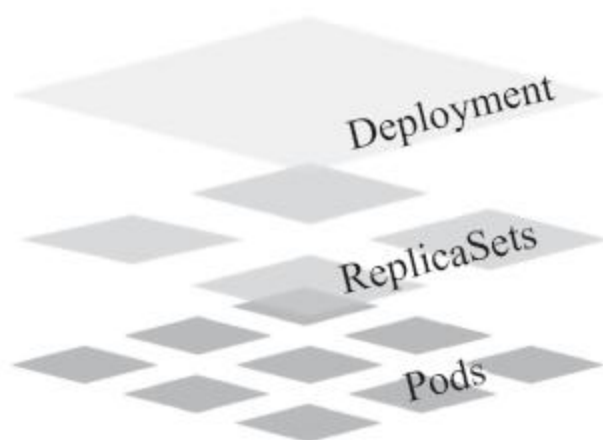


图5-7 Deployment、ReplicaSets和Pods

Deployment控制器资源的主要职责同样是为了保证Pod资源的健康运行，其大部分功能均可通过调用ReplicaSet控制器来实现，同时还增添了部分特性。

- 事件和状态查看：必要时可以查看Deployment对象升级的详细进度和状态。

- 回滚：升级操作完成后发现问题时，支持使用回滚机制将应用返回到前一个或由用户指定的历史记录中的版本上。

- 版本记录：对Deployment对象的每一次操作都予以保存，以供后续可能执行的回滚操作使用。

- 暂停和启动：对于每一次升级，都能够随时暂停和启动。

- 多种自动更新方案：一是Recreate，即重建更新机制，全面停止、删除旧有的Pod后用新版本替代；另一个是RollingUpdate，即滚动

升级机制，逐步替换旧有的Pod至新的版本。

5.3.1 创建Deployment

Deployment是标准的Kubernetes API资源，它建构于ReplicaSet资源之上，于是其spec字段中嵌套使用的字段包含了ReplicaSet控制器支持的replicas、selector、template和minReadySeconds，它也正是利用这些信息完成了其二级资源ReplicaSet对象的创建。下面是一个Deployment控制器资源的配置清单示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
          ports:
            - containerPort: 80
              name: http
```

上面的内容显示出，除了控制器类型和名称之外，它与前面ReplicaSet控制器示例中的内容几乎没有什么不同。下面在集群中创建以了解它的工作方式：

```
~]$ kubectl apply -f myapp-deploy.yaml --record
deployment.apps "myapp-deploy" created
```

“kubectl get deployments”命令可以列出创建的Deployment对象myapp-deploy及其相关的信息。下面显示的字段中，UP-TO-DATE表示已经达到期望状态的Pod副本数量，AVAILABLE则表示当前处于可用状态的应用程序的数量：

```
~]$ kubectl get deployments myapp-deploy
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp-deploy  3         3         3             3           13s
```

Deployment控制器会自动创建相关的ReplicaSet控制器资源，并以“[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]”格式为其命名，其中的hash值由Deployment控制器自动生成。由Deployment创建的ReplicaSet对象会自动使用相同的标签选择器，因此，可使用类似如下的命令查看其相关的信息：

```
~]$ kubectl get replicaset -l app=myapp
NAME                                DESIRED   CURRENT   READY   AGE
myapp-deploy-86b4b8c75d            3         3         3       37s
```

相关的Pod对象的信息可以用相似的命令进行获取。下面的命令结果中，Pod对象的名称遵循ReplicaSet控制器的命名格式，它以ReplicaSet控制器的名称为前缀，后跟5位随机字符：

```
~]$ kubectl get pods -l app=myapp
NAME                                READY     STATUS    RESTARTS   AGE
myapp-deploy-86b4b8c75d-7dtxn      1/1       Running   0          46s
myapp-deploy-86b4b8c75d-hdw9z      1/1       Running   0          46s
myapp-deploy-86b4b8c75d-w4svj      1/1       Running   0          46s
```

由此印证了Deployment借助于ReplicaSet管理Pod资源的机制，于是可以得知，其大部分管理操作与ReplicaSet相同。不过，Deployment也有ReplicaSet所不具有的部分高级功能，这其中最著名的当数其自动滚动更新的机制。

5.3.2 更新策略

如前所述，**ReplicaSet**控制器的应用更新需要手动分成多步并以特定的次序进行，过程繁杂且容易出错，而**Deployment**却只需要由用户指定在**Pod**模板中要改动的内容，例如容器镜像文件的版本，余下的步骤可交由其自动完成。同样，更新应用程序的规模也只需要修改期望的副本数量，余下的事情交给**Deployment**控制器即可。

Deployment控制器详细信息中包含了其更新策略的相关配置信息，如**myapp-deploy**控制器资源“**kubectl describe**”命令中输出的**StrategyType**、**RollingUpdateStrategy**字段等：

```
~]$ kubectl describe deployments myapp-deploy
Name:                myapp-deploy
Namespace:           default
CreationTimestamp:    Fri, 02 Mar 2018 09:57:06 +0800
Labels:              app=deploy-demo
Annotations:         deployment.kubernetes.io/revision=1
Selector:            app=myapp
Replicas:            3 desired | 3 updated | 3 total | 3 available | 0
                    unavailable
StrategyType:        RollingUpdate
MinReadySeconds:     0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
.....
OldReplicaSets:      <none>
NewReplicaSet:       myapp-deploy-86b4b8c75d (3/3 replicas created)
.....
```

Deployment控制器支持两种更新策略：滚动更新（**rolling update**）和重新创建（**recreate**），默认为滚动更新。重新创建更新类似于前文中**ReplicaSet**的第一种更新方式，即首先删除现有的**Pod**对象，而后由控制器基于新模板重新创建出新版本资源对象。通常，只应该在应用的新旧版本不兼容（如依赖的后端数据库的**schema**不同且无法兼容）时运行时才会使用**recreate**策略，因为它会导致应用替换期间暂时不可用，好处在于它不存在中间状态，用户访问到的要么是应用的新版本，要么是旧版本。

滚动升级是默认的更新策略，它在删除一部分旧版本**Pod**资源的同时，补充创建一部分新版本的**Pod**对象进行应用升级，其优势是升级期间，容器中应用提供的服务不会中断，但要求应用程序能够应对新旧

版本同时工作的情形，例如新旧版本兼容同一个数据库方案等。不过，更新操作期间，不同客户端得到的响应内容可能会来自不同版本的应用。

Deployment控制器的滚动更新操作并非在同一个**ReplicaSet**控制器对象下删除并创建**Pod**资源，而是将它们分置于两个不同的控制器之下：旧控制器的**Pod**对象数量不断减少的同时，新控制器的**Pod**对象数量不断增加，直到旧控制器不再拥有**Pod**对象，而新控制器的副本数量变得完全符合期望值为止，如图5-8所示。

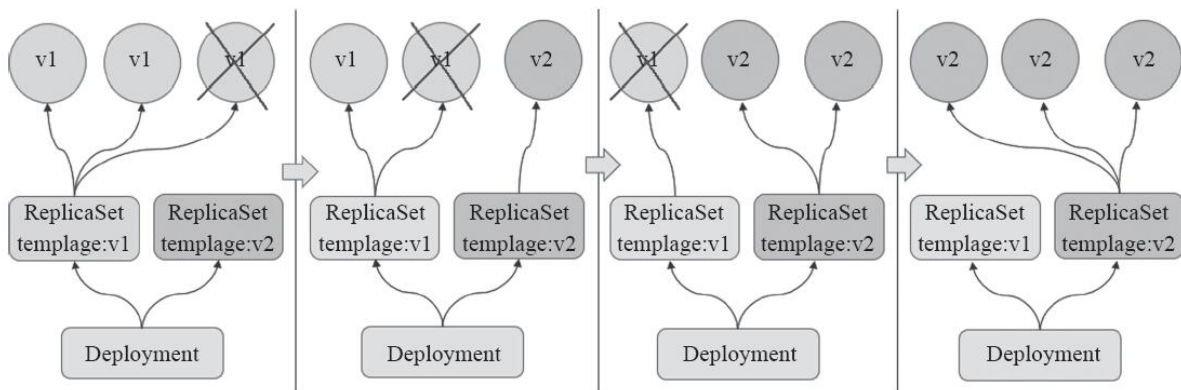


图5-8 Deployment的滚动更新

滚动更新时，应用升级期间还要确保可用的**Pod**对象数量不低于某阈值以确保可以持续处理客户端的服务请求，变动的方式和**Pod**对象的数量范围将通过`spec.strategy.rollingUpdate.maxSurge`和`spec.strategy.rollingUpdate.maxUnavailable`两个属性协同进行定义，它们的功用如图5-9所示。

·**maxSurge**: 指定升级期间存在的总**Pod**对象数量最多可超出期望值的个数，其值可以是0或正整数，也可以是一个期望值的百分比；例如，如果期望值为3，当前的属性值为1，则表示**Pod**对象的总数不能超过4个。

·**maxUnavailable**: 升级期间正常可用的**Pod**副本数（包括新旧版本）最多不能低于期望数值的个数，其值可以是0或正整数，也可以是一个期望值的百分比；默认值为1，该值意味着如果期望值是3，则升级期间至少要有两个**Pod**对象处于正常提供服务的状态。

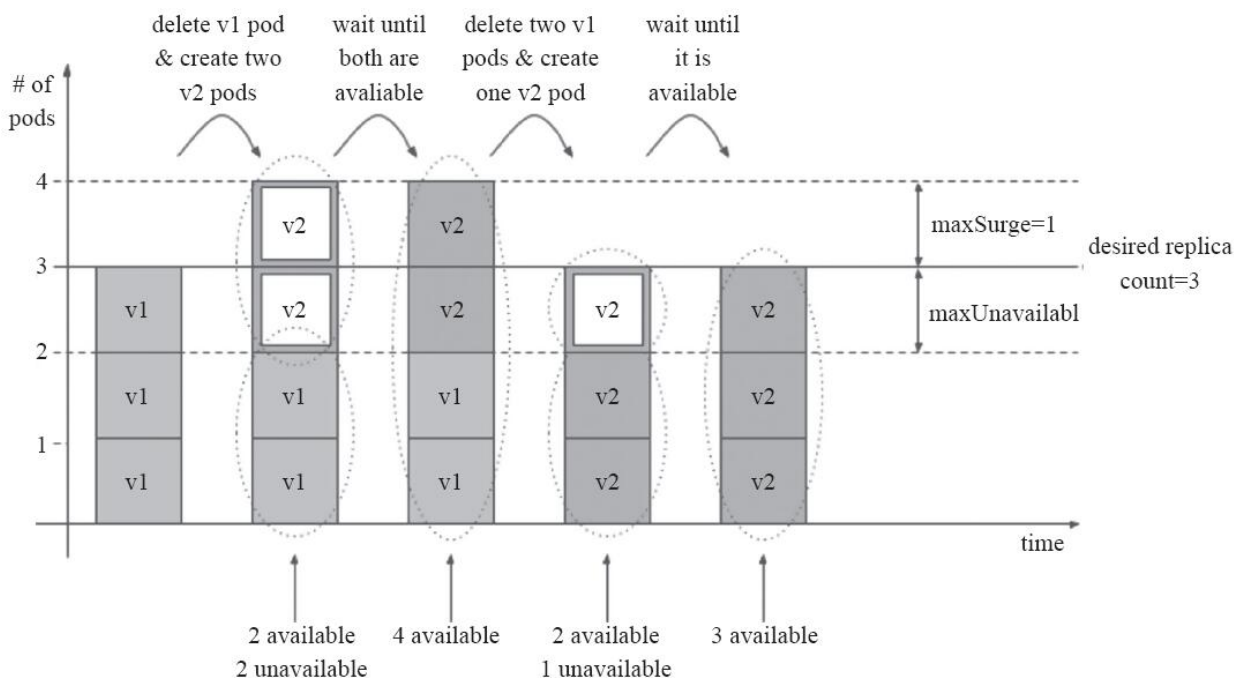


图5-9 `maxSurge`和`maxUnavailable`的作用方式（图片来源：《Kubernetes in action》）



注意 `maxSurge`和`maxUnavailable`属性的值不可同时为0，否则Pod对象的副本数量在符合用户期望的数量后无法做出合理变动以进行滚动更新操作。

配置时，用户还可以使用Deployment控制器的`spec.minReadySeconds`属性来控制应用升级的速度。新旧更替过程中，新创建的Pod对象一旦成功响应就绪探测即被视作可用，而后即可立即开始下一轮的替换操作。而`spec.minReadySeconds`能够定义在新的Pod对象创建后至少要等待多久才会将其视作就绪，在此期间，更新操作会被阻塞。因此，它可以用来让Kubernetes在每次创建出Pod资源后都要等上一段时长后再开始下一轮的更替，这个时间长度的理想值是等到Pod对象中的应用已经可以接受并处理请求流量。事实上，一个精心设计的等待时长和就绪性探测能让Kubernetes系统规避一部分因程序Bug而导致的升级故障。

Deployment控制器也支持用户保留其滚动更新历史中的旧ReplicaSet对象版本，如图5-10所示，这赋予了控制器进行应用回滚的能力：用户可按需回滚到指定的历史版本。控制器可保存的历史版本数量由“`spec.revisionHistoryLimit`”属性进行定义。当然，也只有保存于

revision历史中的ReplicaSet版本可用于回滚，因此，用户要习惯性地在更新操作时指定保留旧版本。

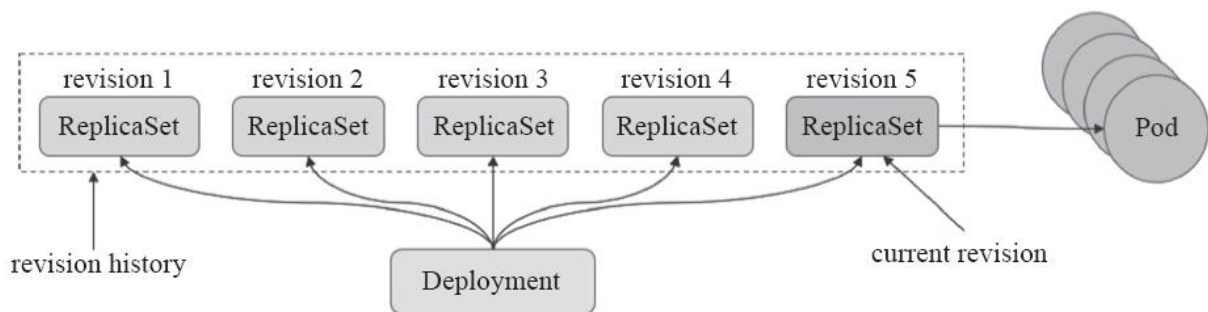


图5-10 Deployment的版本历史记录



注意 为了保存版本升级的历史，需要在创建Deployment对象时于命令中使用“--record”选项。

尽管滚动更新以节约系统资源著称，但它也存在一些劣势。直接改动现有环境，会使系统引入不确定性风险，而且升级过程出现问题后，执行回滚操作也会较为缓慢。有鉴于此，金丝雀部署可能是较为理想的实现方式，当然，如果不考虑系统资源的可用性，那么传统的蓝绿部署也是不错的选择。

5.3.3 升级Deployment

修改Pod模板相关的配置参数便能完成Deployment控制器资源的更新。由于是声明式配置，因此对Deployment控制器资源的修改尤其适合使用apply和patch命令来进行；当然，如果仅是修改容器镜像，“set image”命令更为易用。

接下来通过更新此前创建的Deployment控制器deploy-example来了解更新操作过程的执行细节，为了使得升级过程更易于观测，这里先使用“kubectl patch”命令为其spec.minReadySeconds字段定义一个等待时长，例如5s：

```
~]$ kubectl patch deployments myapp-deploy -p '{"spec": {"minReadySeconds": 5}}'
deployment.extensions "myapp-deploy" patched
```

patch命令的补丁形式为JSON格式，以-p选项指定，上面命令中的'{"spec": {"minReadySeconds": 5}}'表示设置spec.minReadySeconds属性的值。若要改变myapp-deploy中myapp容器的镜像，也可使用patch命令，如'{"spec": {"containers": [{"name": "myapp", "image": "ikubernetes/myapp: v2"}]}'，不过，修改容器镜像有更为简单的专用命令“set image”。



注意 修改Deployment控制器的minReadySeconds、replicas和strategy等字段的值并不会触发Pod资源的更新操作，因为它们不属于模板的内嵌字段，对现存的Pod对象不产生任何影响。

接着，使用“ikubernetes/myapp: v2”镜像文件修改Pod模板中的myapp容器，启动Deployment控制器的滚动更新过程：

```
~]$ kubectl set image deployments myapp-deploy myapp=ikubernetes/myapp:v2
deployment.apps "myapp-deploy" image updated
```

“kubectl rollout status”命令可用于打印滚动更新过程中的状态信息：

```
~]$ kubectl rollout status deployments myapp-deploy
```

另外，还可以使用“`kubectl get deployments--watch`”命令监控其更新过程中Pod对象的变动过程：

```
~]$ kubectl get deployments myapp-deploy --watch
```

滚动更新时，`myapp-deploy`控制器会创建一个新的ReplicaSet控制器对象来管控新版本的Pod对象，升级完成后，旧版本的ReplicaSet会保留在历史记录中，但其此前的管控Pod对象将会被删除。

```
~]$ kubectl get replicaset -l app=myapp
```

NAME	DESIRED	CURRENT	READY	AGE
myapp-deploy-79859f456c	3	3	3	1m
myapp-deploy-86b4b8c75d	0	0	0	7m

`myapp-deploy`控制器管控的Pod资源对象也将随之更新为以新版本ReplicaSet名称“`myapp-deploy-79859f456c`”为前缀的Pod副本，命令结果如下所示：

```
~]$ kubectl get pods -l app=myapp
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-deploy-79859f456c-29rqw	1/1	Running	0	16m
myapp-deploy-79859f456c-fhhwf	1/1	Running	0	15m
myapp-deploy-79859f456c-h4n9d	1/1	Running	0	15m

由于已经处于READY状态，因此上面命令列出的任一Pod资源均可正常向用户提供相关服务，例如，在集群内任一能使用kubectl的节点访问`myapp-deploy-79859f456c-29rqw`中的Web服务，命令如下：

```
~]$ curl $(kubectl get pods myapp-deploy-79859f456c-29rqw -o go-template={{.status.podIP}})
Hello MyApp | Version: v2 | <a href="hostname.html">Pod Name</a>
```

5.3.4 金丝雀发布

Deployment控制器还支持自定义控制更新过程中的滚动节奏，如“暂停”（pause）或“继续”（resume）更新操作，尤其是借助于前文讲到的maxSurge和maxUnavailable属性还能实现更为精巧的过程控制。比如，待第一批新的Pod资源创建完成后立即暂停更新过程，此时，仅存在一小部分新版本的应用，主体部分还是旧的版本。然后，再根据用户特征精心筛选出小部分用户的请求路由至新版本的Pod应用，并持续观察其是否能稳定地按期望的方式运行。确定没有问题后再继续完成余下Pod资源的滚动更新，否则立即回滚更新操作。这便是所谓的金丝雀发布（Canary Release），如图5-11所示。

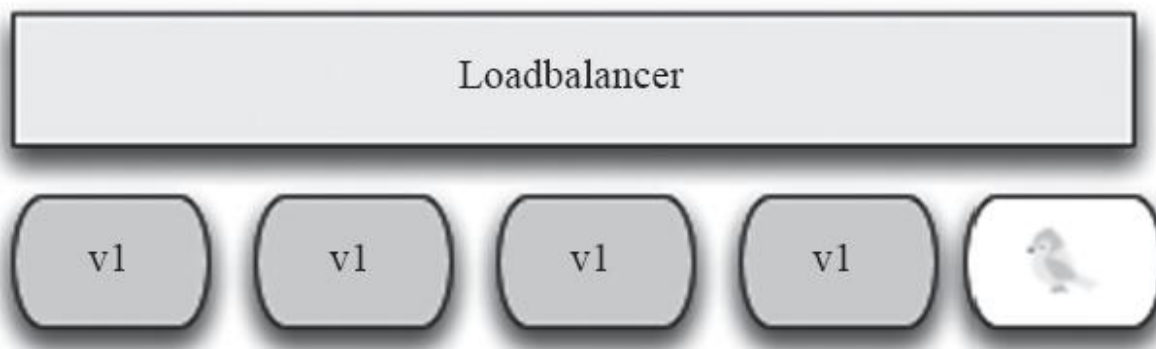


图5-11 金丝雀发布（图片来源：<http://blog.christianposta.com>）

拓展知识：矿井中的金丝雀

17世纪，英国矿井工人发现，金丝雀对瓦斯这种气体十分敏感。空气中哪怕有极其微量的瓦斯气体，金丝雀也会停止歌唱；当瓦斯含量超过一定限度时，人类依旧毫无察觉，而金丝雀却早已毒发身亡。当时在采矿设备相对简陋的条件下，工人们每次下井都会带上一只金丝雀作为瓦斯检测工具，以便在危险状况下紧急撤离。

直接发布新应用版本的在线发布形式中，金丝雀发布是一种较为妥当的方式。不过，这里只涉及其部署操作的相关步骤，发布方式则通常依赖于具体的环境设置。接下来说明如何在Kubernetes上使用Deployment控制器实现金丝雀部署。

为了尽可能地降低对现有系统及其容量的影响，金丝雀发布过程通常建议采用“先添加、再删除，且可用Pod资源对象总数不低于期望值”的方式进行。首次添加的Pod对象数量取决于其接入的第一批请求的规则及单个Pod的承载能力，视具体需求而定，为了能够更简单地说明问题，接下来采用首批添加1个Pod资源的方式。将Deployment控制器的maxSurge属性的值设置为1，并将maxUnavailable属性的值设置为0：

```
~]$ kubectl patch deployments myapp-deploy \
-p '{"spec": {"strategy":{"rollingUpdate": {"maxSurge": 1, "maxUnavailable":
0}}}}'
deployment.extensions "myapp-deploy" patched
```

接下来，启动myapp-deploy控制器的更新过程，在修改相应容器的镜像版本后立即暂停更新进度，它会在启动第一批新版本Pod对象的创建操作之后转为暂停状态。需要注意的是，这里之所以能够在第一批更新启动后就暂停，有赖于此前为maxReadySeconds属性设置的时长，因此用户要在更新命令启动后的此时长指定的时间范围内启动暂停操作，其执行过程如图5-12所示。当然，对kubectl命令来说，也可以直接以“&&”符号在Shell中连接两个命令：

```
~]$ kubectl set image deployments myapp-deploy myapp=ikubernetes/myapp:v3 \
&& kubectl rollout pause deployments myapp-deploy
deployment.apps "myapp-deploy" image updated
deployment.apps "myapp-deploy" paused
```

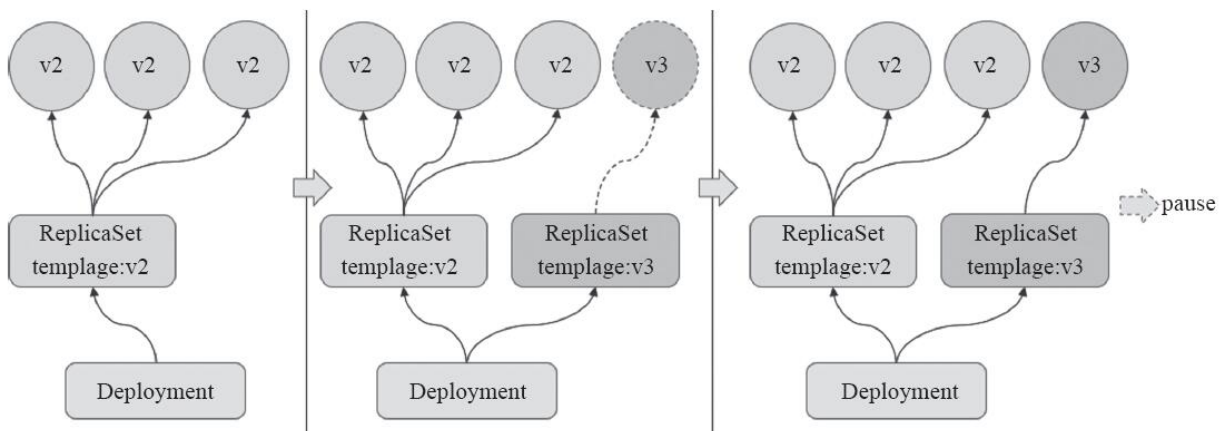


图5-12 暂停Deployment滚动更新

通过其状态查看命令可以看到，在创建完一个新版本的Pod资源后滚动更新操作“暂停”：

```
~]$ kubectl rollout status deployments myapp-deploy
Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
```

相关的Pod列表也可能显示旧版本的ReplicaSet的所有Pod副本仍在正常运行，新版本的ReplicaSet也包含一个Pod副本，但最多不超过期望值1个，myapp-deploy原有的期望值为3，因此总数不超过4个。此时，通过Service或Ingress资源及相关路由策略等设定，即可将一部分用户的流量引入到这些Pod之上进行发布验证。运行一段时间后，如果确认没有问题，即可使用“kubectl rollout resume”命令继续此前的滚动更新过程：

```
~]$ kubectl rollout resume deployments myapp-deploy
deployment.apps "myapp-deploy" resumed
```

“kubectl rollout status”命令监控到滚动更新过程完成后，即可通过myapp-deploy控制器及其ReplicaSet和Pod对象的相关信息来了解其结果状态。

然而，如果“金丝雀”遇险甚至遭遇不幸，那么回滚操作便成了接下来的当紧任务。

5.3.5 回滚Deployment控制器下的应用发布

若因各种原因导致滚动更新无法正常进行，如镜像文件获取失败、“金丝雀”遇险等，则应该将应用回滚到之前的版本，或者回滚到由用户指定的历史记录中的版本。**Deployment**控制器的回滚操作可使用“**kubectl rollout undo**”命令完成，例如，下面的命令可将**myapp-deploy**回滚至此前的版本：

```
~]$ kubectl rollout undo deployments myapp-deploy
deployment.apps "myapp-deploy"
```

等回滚完成后，验证**myapp-deploy**的**ReplicaSet**控制器对象是否已恢复到指定的历史版本以确保其回滚正常完成。在“**kubectl rollout undo**”命令上使用“**--to-revision**”选项指定**revision**号码即可回滚到历史特定版本，例如，假设**myapp-deploy**包含如下的**revision**历史记录：

```
~]$ kubectl rollout history deployments myapp-deploy
deployments "myapp-deploy"
REVISION  CHANGE-CAUSE
1          kubectl patch deployments myapp-deploy --patch={"spec": {"minReady-
Seconds": 5}}
2          kubectl patch deployments myapp-deploy --patch={"spec": {"strategy":
{"rollingUpdate": {"maxSurge": 1, "maxUnavailable": 0}}}}
3          kubectl set image deployments myapp-deploy myapp=ikubernetes/myapp:v3
4          kubectl set image deployments myapp-deploy myapp=ikubernetes/myapp:v4
```

若要回滚到号码为2的**revision**记录，则使用如下命令即可完成：

```
~]$ kubectl rollout undo deployments myapp-deploy --to-revision=2
deployment.apps "myapp-deploy"
```

回滚操作中，其**revision**记录中的信息会发生变动，回滚操作会被当作一次滚动更新追加进历史记录中，而被回滚的条目则会被删除。需要注意的是，如果此前的滚动更新过程处于“暂停”状态，那么回滚操作就需要先将**Pod**模板的版本改回到之前的版本，然后“继续”更新，否则，其将一直处于暂停状态而无法回滚。

5.3.6 扩容和缩容

通过修改`.spec.replicas`即可修改Deployment控制器中Pod资源的副本数量，它将实时作用于控制器并直接生效。Deployment控制器是声明式配置，`replicas`属性的值可直接修改资源配置文件，然后使用“`kubectl apply`”进行应用，也可以使用“`kubectl edit`”对其进行实时修改。而前一种方式能够将修改结果予以长期留存。

另外，“`kubectl scale`”是专用于扩展某些控制器类型的应用规模的命令，包括Deployment和Job等。而Deployment通过ReplicaSet控制其Pod资源，因此扩缩容的方式是相同的，除了命令直接作用的资源对象有所不同之外，这里不再对其进行展开说明。

5.4 DaemonSet控制器

DaemonSet是**Pod**控制器的另一种实现，用于在集群中的全部节点上同时运行一份指定的**Pod**资源副本，后续新加入集群的工作节点也会自动创建一个相关的**Pod**对象，当从集群移除节点时，此类**Pod**对象也将被自动回收而无须重建。管理员也可以使用节点选择器及节点标签指定仅在部分具有特定特征的节点上运行指定的**Pod**对象。

DaemonSet是一种特殊的控制器，它有特定的应用场景，通常运行那些执行系统级操作任务的应用，其应用场景具体如下。

- 运行集群存储的守护进程，如在各个节点上运行glusterd或ceph。
- 在各个节点上运行日志收集守护进程，如fluentd和logstash。
- 在各个节点上运行监控系统的代理守护进程，如Prometheus Node Exporter、collectd、Datadog agent、New Relic agent或Ganglia gmond等。

当然，既然是需要运行于集群内的每个节点或部分节点，于是很多场景中也可以把应用直接运行为工作节点上的系统级守护进程，不过，这样一来就失去了运用**Kubernetes**管理所带来的便捷性。另外，也只有必须将**Pod**对象运行于固定的几个节点并且需要先于其他**Pod**启动时，才有必要使用**DaemonSet**控制器，否则就应该使用**Deployment**控制器。

5.4.1 创建DaemonSet资源对象

DaemonSet控制器的spec字段中嵌套使用的字段同样主要包了前面讲到的**Pod**控制器资源支持的selector、template和minReadySeconds，并且功能和用法基本相同，但它不支持使用replicas，毕竟**DaemonSet**并不是基于期望的副本数来控制**Pod**资源数量，而是基于节点数量，但template是必选字段。

下面的资源清单文件（filebeat-ds.yaml）示例中定义了一个名为filebeat-ds的**DaemonSet**控制器，它将在每个节点上运行一个filebeat进程以收集容器相关的日志数据：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: filebeat-ds
  labels:
    app: filebeat
spec:
  selector:
    matchLabels:
      app: filebeat
  template:
    metadata:
      labels:
        app: filebeat
    name: filebeat
  spec:
    containers:
      - name: filebeat
        image: ikubernetes/filebeat:5.6.5-alpine
        env:
          - name: REDIS_HOST
            value: db.ilinux.io:6379
          - name: LOG_LEVEL
            value: info
```

通过清单文件创建**DaemonSet**资源的命令与其他资源的创建并无二致：

```
~]$ kubectl apply -f filebeat-ds.yaml
daemonset.apps "filebeat-ds" created
```



注意 自Kubernetes 1.8版本起，DaemonSet也必须使用selector来匹配Pod模板中指定的标签，而且它也支持matchLabels和matchExpressions两种标签选择器。

与其他资源对象相同，用户也可以使用“`kubectl describe`”命令查看DaemonSet对象的详细信息。下面命令的结果信息中，Node-Selector字段的值为空，表示它需要运行于集群中的每个节点之上。而当前集群的节点数量为3，因此，其期望的Pod副本数（Desired Number of Nodes Scheduled）为3，而当前也已经成功创建了3个相关的Pod对象：

```
~]$ kubectl describe daemonsets filebeat-ds
Name:          filebeat-ds
Selector:      app=filebeat
Node-Selector: <none>
.....
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Scheduled with Up-to-date Pods: 3
Number of Nodes Scheduled with Available Pods: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
.....
```

根据DaemonSet资源本身的意义，filebeat-ds控制器成功创建的3个Pod对象应该分别运行于集群中的每个节点之上，这一点可以通过如下命令进行验证：

```
~]$ kubectl get pods -l app=filebeat \
-o custom-columns=NAME:metadata.name,NODE:spec.nodeName
NAME          NODE
filebeat-ds-sjrvb  node01.ilinux.io
filebeat-ds-swd47  node03.ilinux.io
filebeat-ds-z7r97  node02.ilinux.io
```

集群中的部分工作节点偶尔也存在需要将Pod对象以单一实例形式运行的情况，例如对于拥有特殊硬件的节点来说，可能会需要为其运行特定的监控代理（agent）程序，等等。其实现方式与前面讲到的Pod资源的节点绑定机制类似，只需要在Pod模板的spec字段中嵌套使用nodeSelector字段，并确保其值定义的标签选择器与部分特定工作节点的标签匹配即可。

5.4.2 更新DaemonSet对象

DaemonSet自Kubernetes 1.6版本起也开始支持更新机制，相关配置定义在spec.update-Strategy嵌套字段中。目前，它支持RollingUpdate（滚动更新）和OnDelete（删除时更新）两种更新策略，滚动更新为默认的更新策略，工作逻辑类似于Deployment控制，不过仅支持使用maxUnavailable属性定义最大不可用Pod资源副本数（默认值为1），而删除时更新的方式则是在删除相应节点的Pod资源后重建并更新为新版本。

例如，将此前创建的filebeat-ds中Pod模板中的容器镜像升级为“ikubernetes/filebeat: 5.6.6-alpine”，使用“kubectl set image”命令即可实现：

```
~]$ kubectl set image daemonsets filebeat-ds filebeat=ikubernetes/filebeat:5.6.6-alpine
daemonset.apps "filebeat-ds" image updated
```

由下面命令的返回结果可以看出，filebeat-ds控制器Pod模板中的容器镜像文件已经完成更新，对滚动更新策略来说，它会自动触发更新操作。用户也可以通过filebeat-ds控制器的详细信息中的Events字段等来了解滚动更新的操作过程。由下面的命令结果可以看出，默认的滚动更新策略是一次删除一个工作节点上的Pod资源，待其新版本Pod资源重建完成后再开始操作另一个工作节点上的Pod资源：

```
~]$ kubectl describe daemonsets filebeat-ds
.....
Events:
  Type      Reason              Age   From                Message
  ----      -
  Normal    SuccessfulDelete    3m    daemonset-controller Deleted pod: filebeat-ds-swd47
  Normal    SuccessfulCreate    3m    daemonset-controller Created pod: filebeat-ds-pnhv1
  Normal    SuccessfulDelete    3m    daemonset-controller Deleted pod: filebeat-ds-z7r97
  Normal    SuccessfulCreate    2m    daemonset-controller Created pod: filebeat-ds-z2wdv
  Normal    SuccessfulDelete    2m    daemonset-controller Deleted pod: filebeat-ds-sjrvb
```

Normal SuccessfulCreate 2m daemonset-controller Created pod: filebeat-
ds-6pvdb

DaemonSet控制器的滚动更新机制也可以借助于**minReadySeconds**字段控制滚动节奏，必要时可以执行暂停和继续操作，因此它也能够设计为金丝雀发布机制。另外，故障的更新操作也可以进行回滚，包括回滚至**revision**历史记录中的任何一个指定的版本。鉴于篇幅，这里不再给出其详细过程，感兴趣的读者可参考**Deployment**控制器的步骤测试其实现。

5.5 Job控制器

与Deployment及DaemonSet控制器管理的守护进程类的服务应用不同的是，Job控制器用于调配Pod对象运行一次性任务，容器中的进程在正常运行结束后不会对其进行重启，而是将Pod对象置于“Completed”（完成）状态。若容器中的进程因错误而终止，则需要依配置确定重启与否，未运行完成的Pod对象因其所在的节点故障而意外终止后会被重新调度。Job控制器的Pod对象的状态转换如图5-13所示。

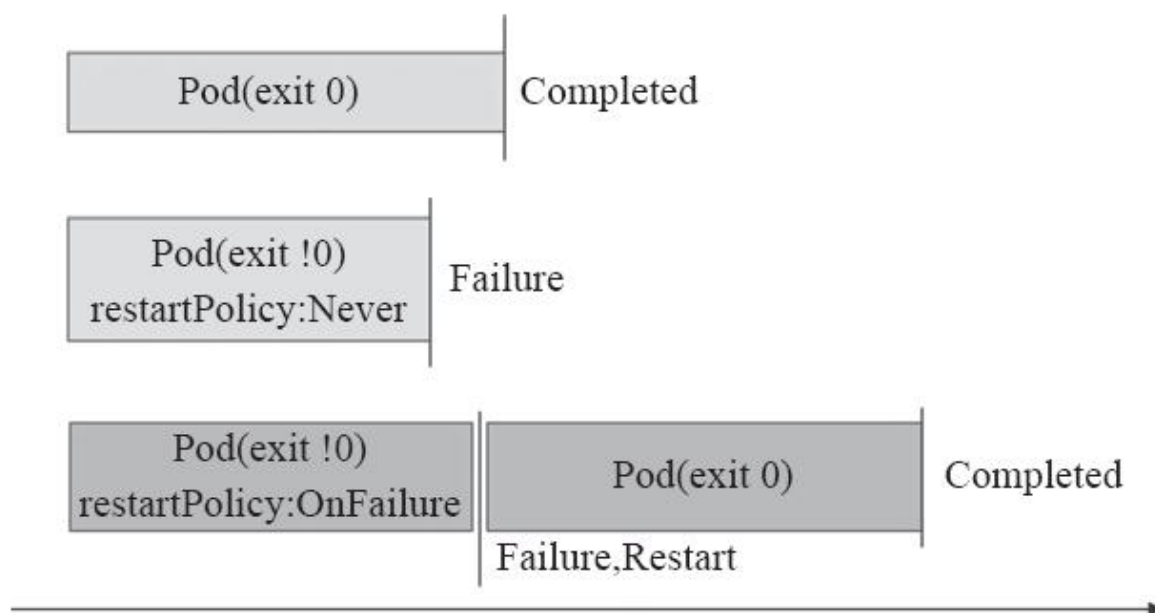


图5-13 Job管理下Pod资源的运行方式

实践中，有的作业任务可能需要运行不止一次，用户可以配置它们以串行或并行的方式运行。总结起来，这种类型的Job控制器对象有两种，具体如下。

- 单工作队列（work queue）的串行式Job：即以多个一次性的作业方式串行执行多次作业，直至满足期望的次数，如图5-14所示；这次Job也可以理解为并行度为1的作业执行方式，在某个时刻仅存在一个Pod资源对象。

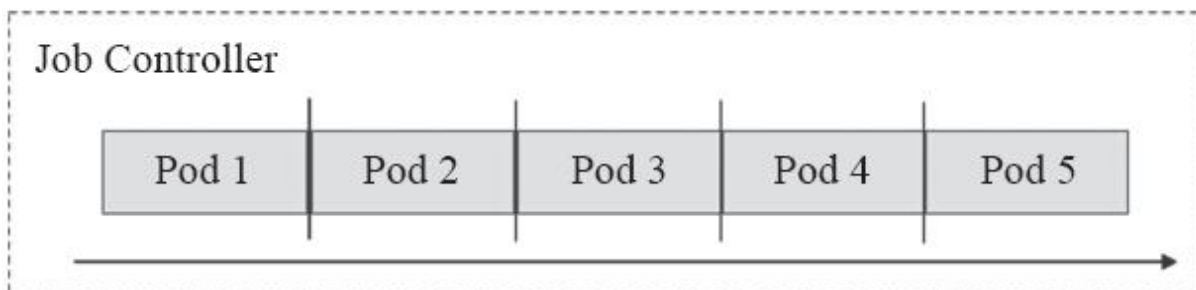


图5-14 串行式多任务

·多工作队列的并行式Job: 这种方式可以设置工作队列数，即作业数，每个队列仅负责运行一个作业，如图5-15a所示；也可以用有限的工作队列运行较多的作业，即工作队列数少于总作业数，相当于运行多个串行作业队列。如图5-15b所示，工作队列数即为同时可运行的Pod资源数。

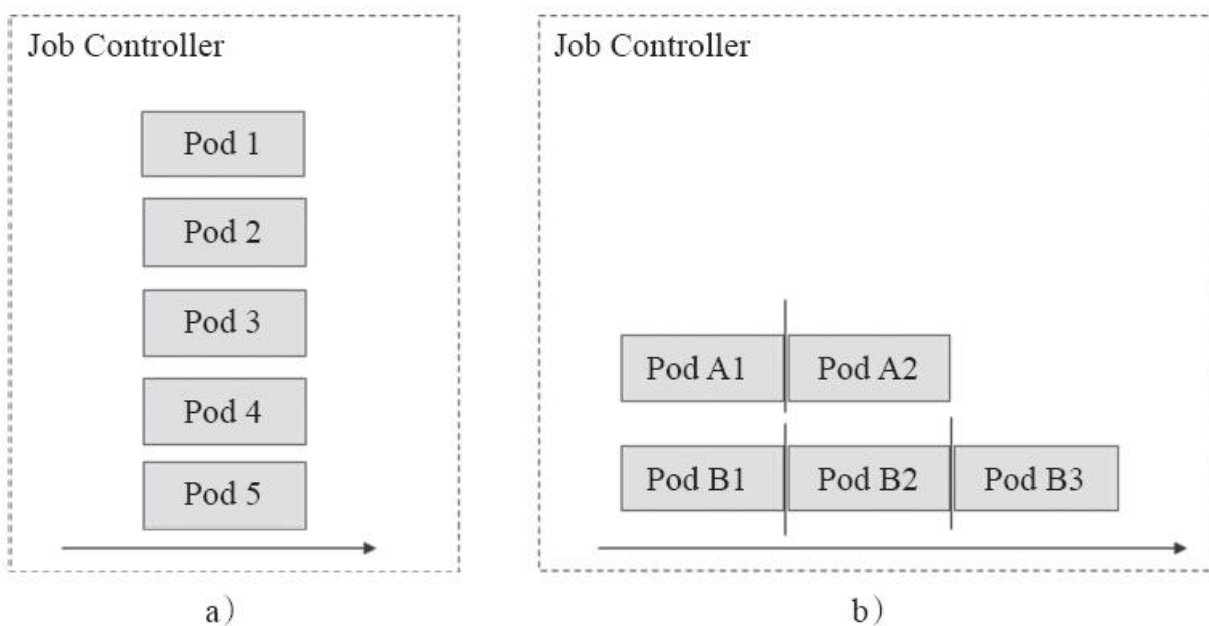


图5-15 多队列并行式多任务

Job控制器常用于管理那些运行一段时间便可“完成”的任务，例如计算或备份操作。

5.5.1 创建Job对象

Job控制器的spec字段内嵌的必要字段仅为template，它的使用方式与Deployment等控制器并无不同。Job会为其Pod对象自动添加“job-name=JOB_NAME”和“controller-uid=UID”标签，并使用标签选择器完成对controller-uid标签的关联。需要注意的是，Job位于API群组“batch/v1”之内。下面的资源清单文件（job-example.yaml）中定义了一个Job控制器：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  template:
    spec:
      containers:
      - name: myjob
        image: alpine
        command: ["/bin/sh", "-c", "sleep 120"]
      restartPolicy: Never
```



注意 Pod模板中的spec.restartPolicy默认为“Always”，这对Job控制器来说并不适用，因此必须在Pod模板中显式设定restartPolicy属性的值为“Never”或“OnFailure”。

使用“kubectl create”或“kubectl apply”命令完成创建后即可查看相关的任务状态，DESIRED字段表示期望并行运行的Pod资源数量，而SUCCESSFUL则表示成功完成的Job数：

```
~]$ kubectl get jobs job-example
NAME           DESIRED  SUCCESSFUL  AGE
job-example    1        0           7s
```

相关的Pod资源能够以Job控制器名称为标签进行匹配：

```
~]$ kubectl get pods -l job-name=job-example
NAME                READY  STATUS   RESTARTS  AGE
job-example-6c5g8   1/1    Running  0         20s
```

其详细信息中可显示所使用的标签选择器及匹配的Pod资源的标签，具体如下：

```
~]$ kubectl describe jobs job-example
Name:          job-example
Namespace:     default
Selector:      controller-uid=48ae496f-1e81-11e8-9267-000c29ab0f5b
Labels:        controller-uid=48ae496f-1e81-11e8-9267-000c29ab0f5b
               job-name=job-example
Annotations:   <none>
Parallelism:   1
Completions:   1
.....
```

两分钟后，待sleep命令执行完成并成功退出后，Pod资源即转换为Completed状态，并且不会于“kubectl get pods”命令中出现，除非为其使用选项“--show-all”或简单格式的“-a”：

```
~]$ kubectl get pods -l job-name=job-example -a
NAME                READY    STATUS    RESTARTS   AGE
job-example-6c5g8   0/1     Completed  0          3m
```

此时，如果使用“kubectl get jobs”显示job-example的相关信息，那么其SUCCESSFUL字段的数字就不再为“0”。

5.5.2 并行式Job

将并行度属性`.spec.parallelism`的值设置为1，并设置总任务数`.spec.completion`属性便能够让Job控制器以串行方式运行多任务。下面是一个串行运行5次任务的Job控制器示例：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-multi
spec:
  completions: 5
  template:
    spec:
      containers:
      - name: myjob
        image: alpine
        command: ["/bin/sh", "-c", "sleep 20"]
        restartPolicy: OnFailure
```

在创建之后或者创建之前，可以于另一终端启动Pod资源的列出命令“`kubectrl get pods-l job-name=job-multi--watch`”来监控其变动，以了解其执行过程。

`.spec.parallelism`能够定义作业执行的并行度，将其设置为2或者以上的值即可实现并行多队列作业运行。同时，如果`.spec.completions`使用的是默认值1，则表示并行度即作业总数，如图5-15a所示；而如果将`.spec.completions`属性值设置为大于`.spec.parallelism`的属性值，则表示使用多队列串行任务作业模式，如图5-15b所示。例如，某Job控制器配置中的`spec`字段嵌套了如下属性，表示以2个队列并行的方式，总共运行5次的作业：

```
spec:
  completions: 5
  parallelism: 2
```

5.5.3 Job扩容

Job控制器的.spec.parallelism定义的并行度表示同时运行的Pod对象数，此属性值支持运行时调整从而改变其队列总数，实现扩容和缩容。使用的命令与此前的Deployment对象相同，即“`kubectl scale --replicas`”命令，例如在其运行过程中（未完成之前）将job-multi的并行度扩展为两路：

```
~]$ kubectl scale jobs job-multi --replicas=2
job.apps "job-multi" scaled
```

执行命令后可以看到，其同时运行的Pod对象副本数量立即扩展到了两个：

```
~]$ kubectl get pods -l job-name=job-multi
```

NAME	READY	STATUS	RESTARTS	AGE
job-multi-c26rh	0/1	ContainerCreating	0	2s
job-multi-tfj9k	1/1	Running	0	26s

根据工作节点及其资源可用量，适度提高Job的并行度，能够大大提升其完成效率，缩短运行时间。

5.5.4 删除Job

Job控制器待其Pod资源运行完成后，将不再占用系统资源。用户可按需保留或使用资源删除命令将其删除。不过，如果某Job控制器的容器应用总是无法正常结束运行，而其restartPolicy又定为了重启，则它可能会一直处于不停地重启和错误的循环当中。所幸的是，Job控制器提供了两个属性用于抑制这种情况的发生，具体如下。

`.spec.activeDeadlineSeconds<integer>`：Job的deadline，用于为其指定最大活动时间长度，超出此时长的作业将被终止。

`.spec.backoffLimit<integer>`：将作业标记为失败状态之前的重试次数，默认值为6。

例如，下面的配置片断表示其失败重试的次数为5，并且如果超出100秒的时间仍未运行完成，那么其将被终止：

```
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 100
```

5.6 CronJob控制器

CronJob控制器用于管理**Job**控制器资源的运行时间。**Job**控制器定义的作业任务在其控制器资源创建之后便会立即执行，但**CronJob**可以类似于**Linux**操作系统的周期性任务作业计划（**crontab**）的方式控制其运行的时间点及重复运行的方式，具体如下。

- 在未来某时间点运行作业一次。
- 在指定的时间点重复运行作业。

CronJob对象支持使用的时间格式类似于**Crontab**，略有不同的是，**CronJob**控制器在指定的时间点时，“”和“*”的意义相同，都表示任何可用的有效值。

5.6.1 创建CronJob对象

CronJob控制器的spec字段可嵌套使用以下字段。

·`jobTemplate<Object>`: Job控制器模板，用于为CronJob控制器生成Job对象；必选字段。

·`schedule<string>`: Cron格式的作业调度运行时间点；必选字段。

·`concurrencyPolicy<string>`: 并发执行策略，可用值有“Allow”（允许）、“Forbid”（禁止）和“Replace”（替换），用于定义前一次作业运行尚未完成时是否以及如何运行后一次的作业。

·`failedJobHistoryLimit<integer>`: 为失败的任务执行保留的历史记录数，默认为1。

·`successfulJobsHistoryLimit<integer>`: 为成功的任务执行保留的历史记录数，默认为3。

·`startingDeadlineSeconds<integer>`: 因各种原因缺乏执行作业的时间点所导致的启动作业错误的超时时长，会被记入错误历史记录。

·`suspend<boolean>`: 是否挂起后续的任务执行，默认为false，对运行中的作业不会产生影响。

下面是一个定义在资源清单文件（`cronjob-example.yaml`）中的CronJob资源对象示例，它每隔2分钟运行一次由jobTemplate定义的简单任务：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
  labels:
    app: mycronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    metadata:
      labels:
```

```
    app: mycronjob-jobs
spec:
  parallelism: 2
  template:
    spec:
      containers:
      - name: myjob
        image: alpine
        command:
        - /bin/sh
        - -c
        - date; echo Hello from the Kubernetes cluster; sleep 10
      restartPolicy: OnFailure
```

运行资源创建命令创建上述CronJob资源对象，而后再通过资源对象的相关信息了解运行状态。下面命令结果中的SCHEDULE是指其调度时间点，SUSPEND表示后续任务是否处于挂起状态，即暂停任务的调度和运行，ACTIVE表示活动状态的Job对象的数量，而LAST SCHEDULE则表示上次调度运行至此刻的时长：

```
~]$ kubectl get cronjobs cronjob-example
NAME                SCHEDULE    SUSPEND    ACTIVE    LAST SCHEDULE    AGE
cronjob-example     */2 * * * * False       1          37s        1m
```



提示 自Kubernetes 1.8起，CronJob资源所在的API资源组从batch/v2alpha1移至batch/v1beta1中，并且查看其资源格式时也要使用--api-version选项指定其所在的资源组，即“kubectl explain cronjob--api-version='batch/v1beta1'”。

5.6.2 CronJob的控制机制

CronJob控制器是一个更高级别的资源，它以Job控制器资源为其管控对象，并借助它管理Pod资源对象。因此，要使用类似如下命令来查看某CronJob控制器创建的Job资源对象，其中的标签“mycronjob-jobs”是在创建cronjob-example时为其指定。不过，只有相关的Job对象被调度执行时，此命令才能将其正常列出。可列出的Job对象的数量取决于CronJob资源的.spec.successfulJobsHistoryLimit的属性值，默认为3。

```
~]$ kubectl get jobs -l app=mycronjob-jobs
```

NAME	DESIRED	SUCCESSFUL	AGE
cronjob-example-1520057880	<none>	2	5m
cronjob-example-1520058000	<none>	2	3m
cronjob-example-1520058120	<none>	2	1m

如果作业重复执行时指定的时间点较近，而作业执行时长（普遍或偶尔）跨过了其两次执行的时间长度，则会出现两个Job对象同时存在的情形。有些Job对象可能会存在无法或不能同时运行的情况，这个时候就要通过.spec.concurrencyPolicy属性控制作业并存的机制，其默认值为“Allow”，即允许前后Job，甚至属于同一个CronJob的更多Job同时运行。其他两个可用值中，“Forbid”用于禁止前后两个Job同时运行，如果前一个尚未结束，后一个则不予启动（跳过），“Replace”用于让后一个Job取代前一个，即终止前一个并启动后一个。

5.7 ReplicationController

ReplicationController（简称rc或RC）是Kubernetes较早实现的Pod控制器，用于确保Pod资源的不间断运行。不过，Kubernetes后来设计了ReplicaSet及其更高一级的控制器Deployment来取代ReplicationController，并表示在后来的版本中可能会废弃RC。因此，这里不再对ReplicationController做过多的介绍。事实上，它的使用方式与ReplicaSet相同，一旦用到时，绝大多数操作都可以迁移使用，感兴趣的读者可以自行测试。

5.8 Pod中断预算

尽管Deployment或ReplicaSet一类的控制器能够确保相应Pod对象的副本数量不断逼近期望的数量，但它却无法保证在某一时刻一定会存在指定数量或比例的Pod对象，然而这种需求在某些强调服务可用性的场景中却是必备的。于是，Kubernetes自1.4版本起开始引入Pod中断预算（PodDisruptionBudget，简称PDB）类型的资源，用于为那些自愿的（Voluntary）中断做好预算方案（Budget），限制可自愿中断的最大Pod副本数或确保最少可用的Pod副本数，以确保服务的高可用性。

Pod对象会一直存在，除非有意将其销毁，或者出现了不可避免的硬件或系统软件错误。**非自愿中断**是指那些由不可控外界因素导致的Pod中断退出操作，例如，硬件或系统内核故障、网络故障以及节点资源不足导致Pod对象被驱逐等；而那些由用户特地执行的管理操作导致的Pod中断则称为“**自愿中断**”，例如排空节点、人为删除Pod对象、由更新操作触发的Pod对象重建等。部署在Kubernetes的每个应用程序都可以创建一个对应的PDB对象以限制自愿中断时最大可以中断的副本数或者最少应该保持可用的副本数，从而保证应用自身的高可用性。

PDB资源可以用来保护由控制器管理的应用，此时几乎必然意味着PDB使用等同于相关控制器对象的标签选择器以精确关联至目标Pod对象，支持的控制器类型包括Deployment、ReplicaSet和StatefulSet等。同时，PDB对象也可以用来保护那些纯粹是由定制的标签选择器自由选择的Pod对象。

定义PDB资源时，其spec字段主要嵌套使用以下三个字段。

- selector<Object>: 当前PDB对象使用的标签选择器，一般是与相关的Pod控制器使用同一个选择器。

- minAvailable<string>: Pod自愿中断的场景中，至少要保证可用的Pod对象数量或比例，要阻止任何Pod对象发生自愿中断，可将其设置为100%。

·**maxUnavailable<string>**: Pod自愿中断的场景中，最多可转换为不可用状态的Pod对象数量或比例，0值意味着不允许Pod对象进行自愿中断；此字段与**minAvailable**互斥。

下面的示例定义了一个PDB对象，它对5.3.1节中由Deployment控制器myapp-deploy创建的Pod对象设置了Pod中断预算，要求其最少可用的Pod对象数量为2个：

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: myapp-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: myapp
```

PDB资源对象创建完成后，在它的简要信息输出中也标明了最少可用的Pod对象个数，以及允许中断的Pod对象个数：

```
~]$ kubectl get pdb
NAME             MIN AVAILABLE  MAX UNAVAILABLE  ALLOWED DISRUPTIONS  AGE
myapp-pdb        2              N/A              1                    24s
```

接下来可通过命令手动删除myapp-deploy控制器下的所有Pod对象模拟自愿中断过程，并监控各Pod对象被终止的过程来验证PDB资源对象的控制功效。

5.9 本章小结

本章主要讲解了Kubernetes的Pod控制器，它们是“工作负载”类资源的核心组成部分，是基于Kubernetes运行应用的最重要的资源类型之一，具体如下。

- 工作负载类型的控制器根据业务需求管控Pod资源的生命周期。

- ReplicaSet可以确保守护进程型的Pod资源始终具有精确的、处于运行状态的副本数量，并支持Pod规模的伸缩机制；它是新一代的ReplicationController控制器，不过用户通常不应该直接使用ReplicaSet，而是要使用Deployment。

- Deployment是建构在ReplicaSet上的更加抽象的工作负载型控制器，支持多种更新策略及发布机制。

- Job控制器能够控制相应的作业任务得以正常完成并退出，支持并行式多任务。

- CronJob控制器用于控制周期性作业任务，其功能类似于Linux操作系统上的Crontab。

- PodDisruptionBudget资源对象为Kubernetes系统上的容器化应用提供了高可用能力。

第6章 Service和Ingress

运行于Pod中的部分容器化应用是向客户端提供服务的守护进程，例如，`nginx`、`tomcat`和`etcd`等，它们受控于控制器资源对象，存在生命周期，在自愿或非自愿中断后只能被重构的新Pod对象所取代，属于非可再生类的组件。于是，在动态、弹性的管理模型下，**Service**资源用于为此类Pod对象提供一个固定、统一的访问接口及负载均衡的能力，并支持借助于新一代DNS系统的服务发现功能，解决客户端发现并访问容器化应用的难题。

然而，**Service**及Pod对象的IP地址都仅在Kubernetes集群内可达，它们无法接入集群外部的访问流量。解决此类问题的办法中，除了在单一节点上做端口暴露（`hostPort`）及让Pod资源共享使用工作节点的网络名称空间（`hostNetwork`）之外，更推荐用户使用的是**NodePort**或**LoadBalancer**类型的**Service**资源，或者是有着七层负载均衡能力的**Ingress**资源。

6.1 Service资源及其实现模型

Service是Kubernetes的核心资源类型之一，通常可看作微服务的一种实现。事实上它是一种抽象：通过规则定义出由多个**Pod**对象组合而成的逻辑集合，以及访问这组**Pod**的策略。**Service**关联**Pod**资源的规则要借助于标签选择器来完成，这一点类似于第5章讲到的**Pod**控制器。

6.1.1 Service资源概述

由Deployment等控制器管理的Pod对象中断后会由新建的资源对象所取代，而扩缩容后的应用则会带来Pod对象群体的变动，随之变化的还有Pod的IP地址访问接口等，这也是编排系统之上的应用程序必然要面临的问题。例如，当图6-1中的Nginx Pod作为客户端访问tomcat Pod中的应用时，IP的变动或应用规模的缩减会导致客户端访问错误，而Pod规模的扩容又会使得客户端无法有效地使用新增的Pod对象，从而影响达成规模扩展之目的。为此，Kubernetes特地设计了Service资源来解决此类问题。

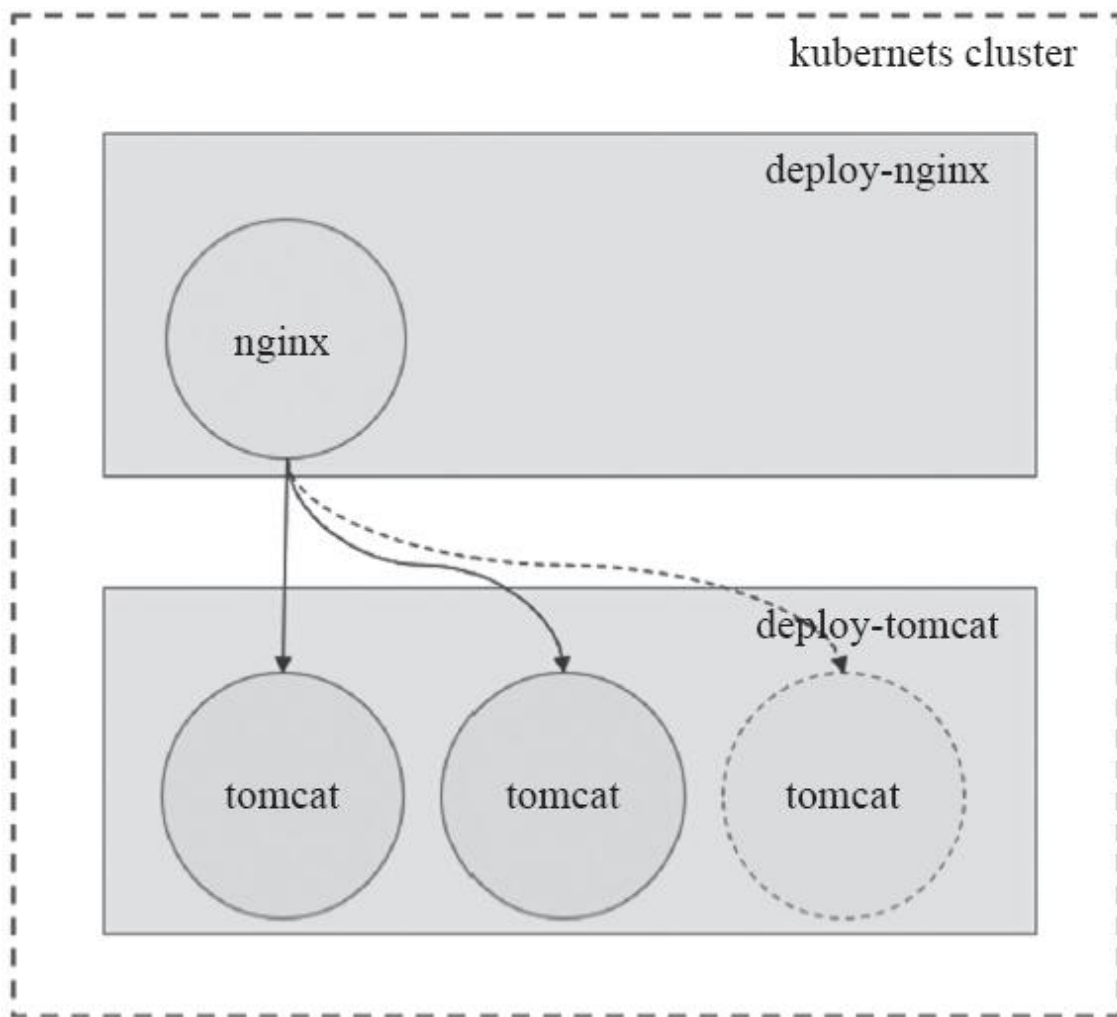


图6-1 Pod及其客户端示例

Service资源基于标签选择器将一组Pod定义成一个逻辑组合，并通过自己的IP地址和端口调度代理请求至组内的Pod对象之上，如图6-2所示，它向客户端隐藏了真实的、处理用户请求的Pod资源，使得客户端的请求看上去就像是由Service直接处理并进行响应的一样。

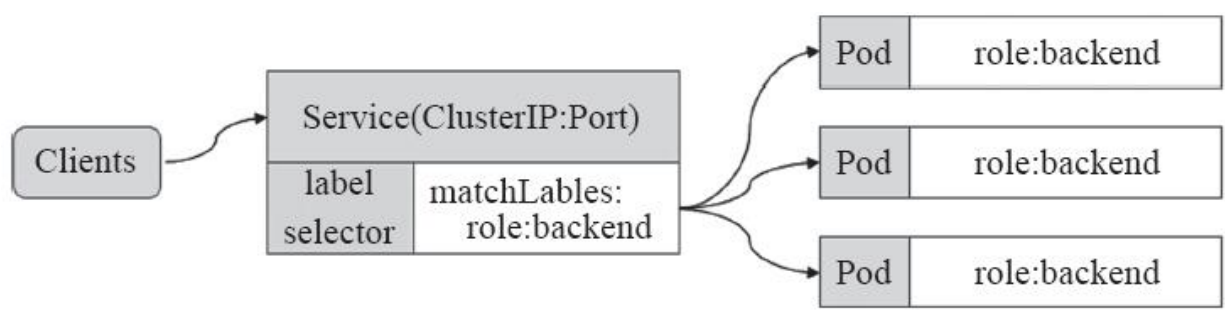


图6-2 Kubernetes Service资源模型示意图

Service对象的IP地址也称为Cluster IP，它位于为Kubernetes集群配置指定专用IP地址的范围之内，而且是一种虚拟IP地址，它在Service对象创建后即保持不变，并且能够被同一集群中的Pod资源所访问。Service端口用于接收客户端请求并将其转发至其后端的Pod中应用的相应端口之上，因此，这种代理机制也称为“端口代理”（port proxy）或四层代理，它工作于TCP/IP协议栈的传输层。

通过其标签选择器匹配到的后端Pod资源不止一个时，Service资源能够以负载均衡的方式进行流量调度，实现了请求流量的分发机制。Service与Pod对象之间的关联关系通过标签选择器以松耦合的方式建立，它可以先于Pod对象创建而不会发生错误，于是，创建Service与Pod资源的任务可由不同的用户分别完成，例如，服务架构的设计和创建由运维工程师进行，而填充其实现的Pod资源的任务则可交由开发者进行。Service、控制器与Pod之间的关系如图6-3所示。

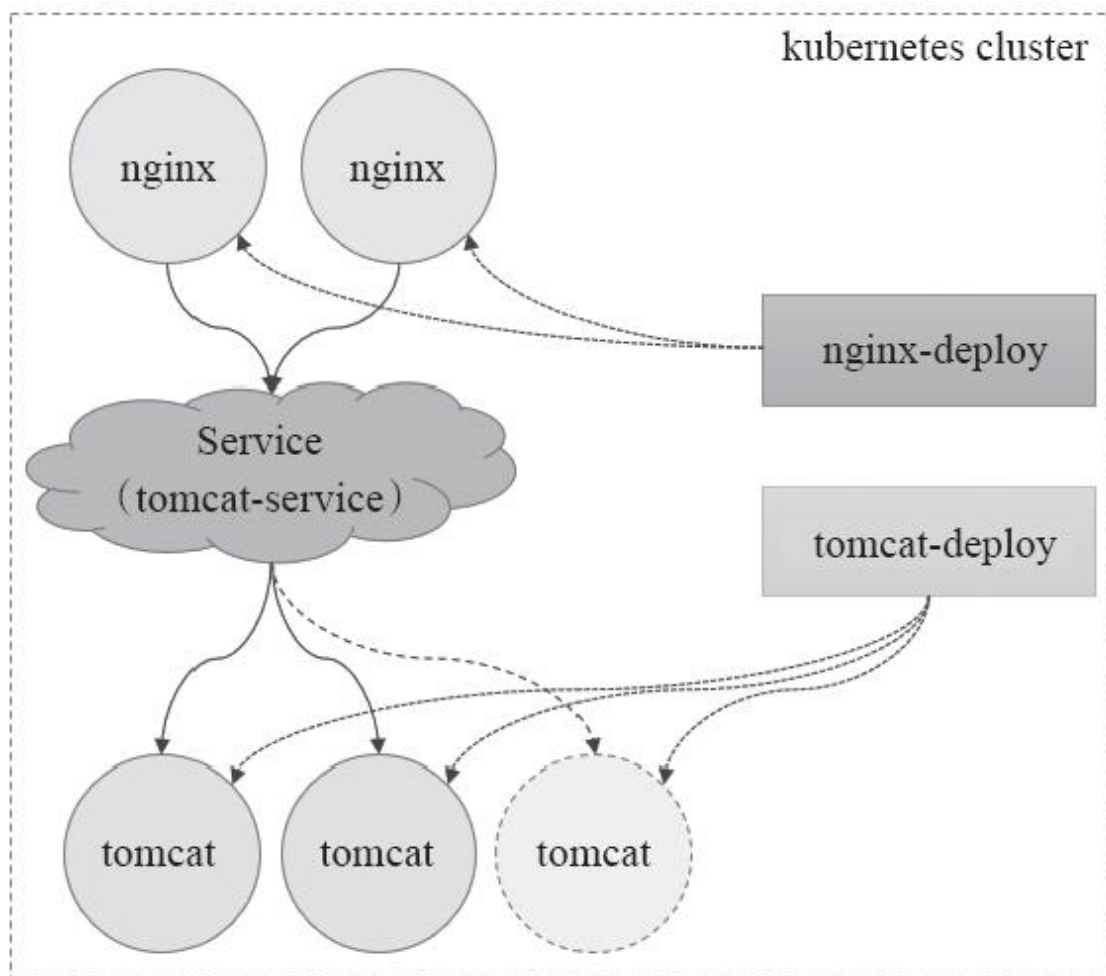


图6-3 Service、控制器与Pod

Service资源会通过API Server持续监视着（watch）标签选择器匹配到的后端Pod对象，并实时跟踪各对象的变动，例如，IP地址变动、对象增加或减少等。不过，需要特别说明的是，Service并不直接链接至Pod对象，它们之间还有一个中间层—Endpoints资源对象，它是一个由IP地址和端口组成的列表，这些IP地址和端口则来自于由Service的标签选择器匹配到的Pod资源。这也是很多场景中会使用“Service的后端端点”（Endpoints）这一术语的原因。默认情况下，创建Service资源对象时，其关联的Endpoints对象会自动创建。

6.1.2 虚拟IP和服务代理

简单来讲，一个Service对象就是工作节点上的一些iptables或ipvs规则，用于将到达Service对象IP地址的流量调度转发至相应的Endpoints对象指向的IP地址和端口之上。工作于每个工作节点的kube-proxy组件通过API Server持续监控着各Service及与其关联的Pod对象，并将其创建或变动实时反映至当前工作节点上相应的iptables或ipvs规则上。客户端、Service及其Pod对象的关系如图6-4所示。



提示 Netfilter是Linux内核中用于管理网络报文的框架，它具有网络地址转换（NAT）、报文改动和报文过滤等防火墙功能，用户借助于用户空间的iptables等工具可按需自由定制规则使用其各项功能。ipvs是借助于Netfilter实现的网络请求报文调度框架，支持rr、wrr、lc、wlc、sh、sed和nq等十余种调度算法，用户空间的命令行工具是ipvsadm，用于管理工作于ipvs之上的调度规则。

Service IP事实上是用于生成iptables或ipvs规则时使用的IP地址，它仅用于实现Kubernetes集群网络的内部通信，并且仅能够将规则中定义的转发服务的请求作为目标地址予以响应，这也是它被称为虚拟IP的原因之一。kube-proxy将请求代理至相应端点的方式有三种：userspace（用户空间）、iptables和ipvs。

1.userspace代理模型

此处的userspace是指Linux操作系统的用户空间。这种模型中，kube-proxy负责跟踪API Server上Service和Endpoints对象的变动（创建或移除），并据此调整Service资源的定义。对于每个Service对象，它会随机打开一个本地端口（运行于用户空间的kube-proxy进程负责监听），任何到达此代理端口的连接请求都将被代理至当前Service资源后端的各Pod对象上，至于会挑中哪个Pod对象则取决于当前Service资源的调度方式，默认的调度算法是轮询（round-robin），其工作逻辑如图6-5所示。另外，此类的Service对象还会创建iptables规则以捕获任何到达ClusterIP和端口的流量。在Kubernetes 1.1版本之前，userspace是默认的代理模型。

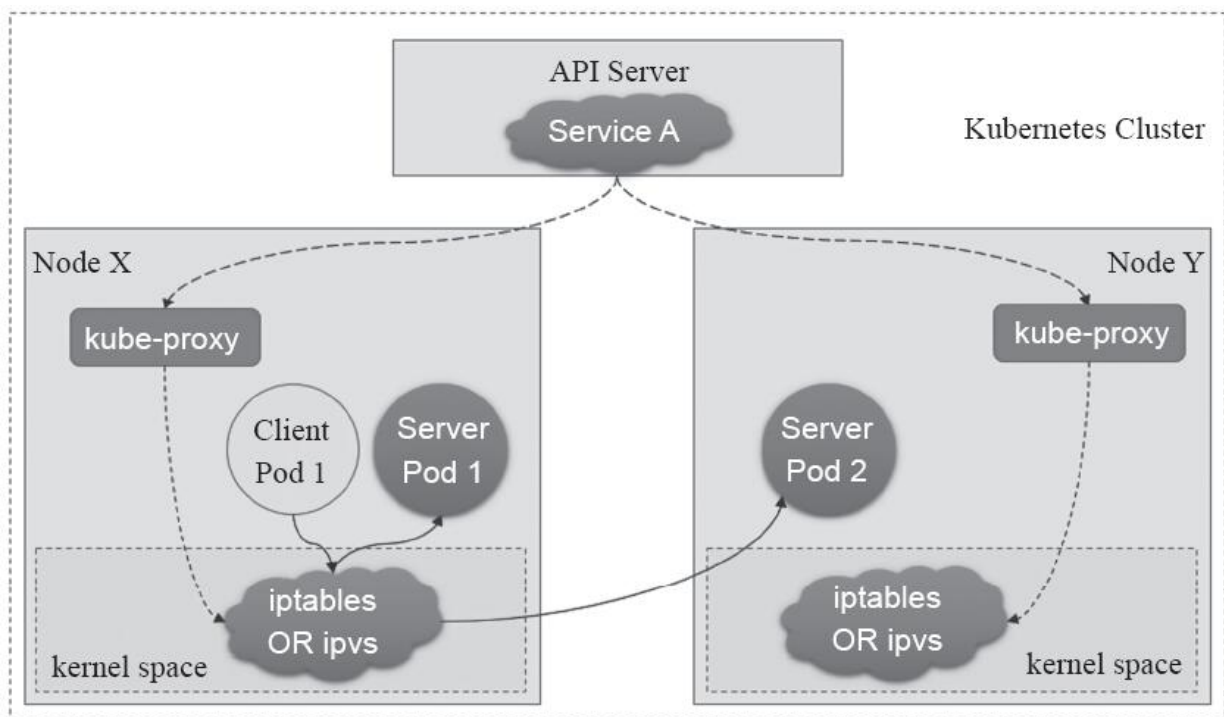


图6-4 kube-proxy和服务

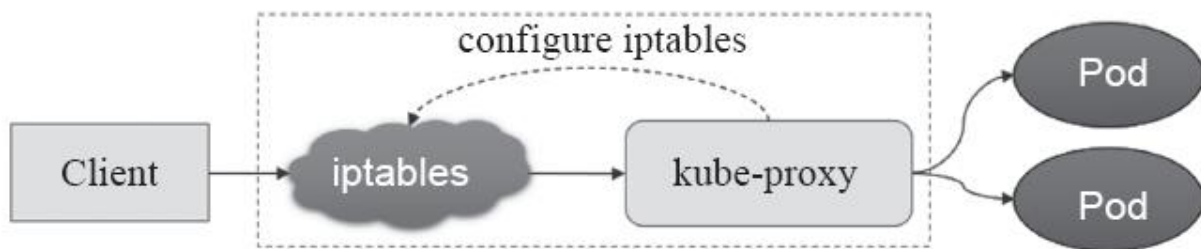


图6-5 userspace代理模型

这种代理模型中，请求流量到达内核空间后经由套接字送往用户空间的**kube-proxy**，而后再由它送回内核空间，并调度至后端**Pod**。这种方式中，请求在内核空间和用户空间来回转发必然会导致效率不高。

2.iptables代理模型

同前一种代理模型类似，**iptables**代理模型中，**kube-proxy**负责跟踪API Server上**Service**和**Endpoints**对象的变动（创建或移除），并据此做出**Service**资源定义的变动。同时，对于每个**Service**，它都会创建**iptables**规则直接捕获到达**ClusterIP**和**Port**的流量，并将其重定向至当前**Service**的后端，如图6-6所示。对于每个**Endpoints**对象，**Service**资源会

为其创建iptables规则并关联至挑选的后端Pod资源，默认算法是随机调度（random）。iptables代理模式由Kubernetes 1.1版本引入，并自1.2版开始成为默认的类型。

在创建Service资源时，集群中每个节点上的kube-proxy都会收到通知并将其定义为当前节点上的iptables规则，用于转发工作接口接收到的与此Service资源的ClusterIP和端口的相关流量。客户端发来的请求被相关的iptables规则进行调度和目标地址转换（DNAT）后再转发至集群内的Pod对象之上。

相对于用户空间模型来说，iptables模型无须将流量在用户空间和内核空间来回切换，因而更加高效和可靠。不过，其缺点是iptables代理模型不会在被挑中的后端Pod资源无响应时自动进行重定向，而userspace模型则可以。

3.ipvs代理模型

Kubernetes自1.9-alpha版本起引入了ipvs代理模型，且自1.11版本起成为默认设置。此种模型中，kube-proxy跟踪API Server上Service和Endpoints对象的变动，据此来调用netlink接口创建ipvs规则，并确保与API Server中的变动保持同步，如图6-7所示。它与iptables规则的不同之处仅在于其请求流量的调度功能由ipvs实现，余下的其他功能仍由iptables完成。

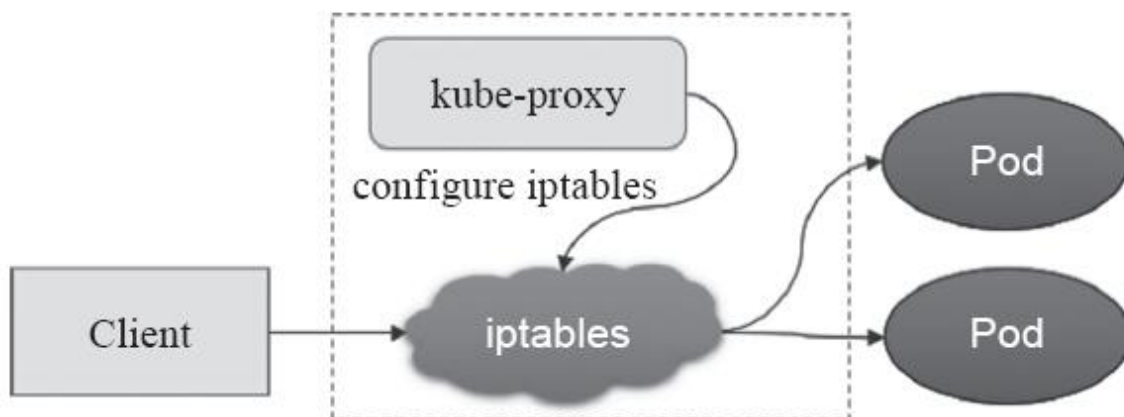


图 6-6 iptables 代理模型

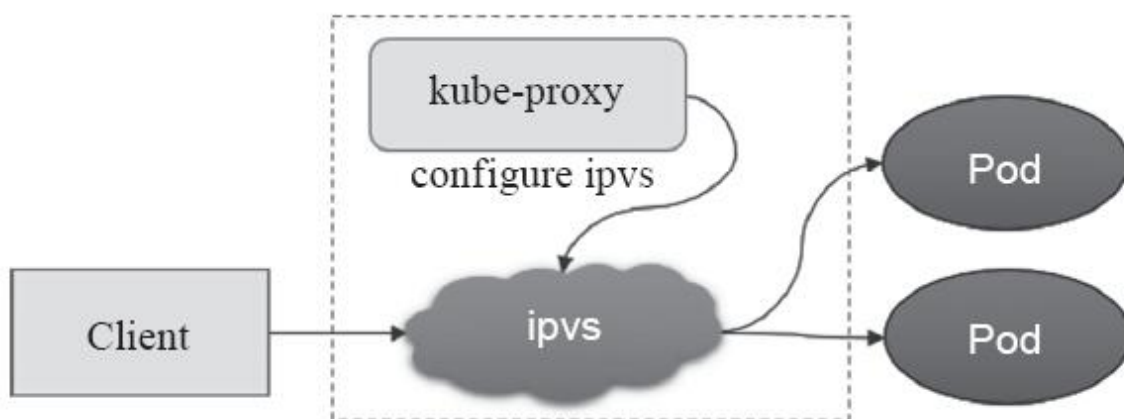


图6-7 ipvs代理模型

类似于iptables模型，ipvs构建于netfilter的钩子函数之上，但它使用hash表作为底层数据结构并工作于内核空间，因此具有流量转发速度快、规则同步性能好的特性。另外，ipvs支持众多调度算法，例如rr、lc、dh、sh、sed和nq等。

6.2 Service资源的基础应用

Service资源本身并不提供任何服务，真正处理并响应客户端请求的是后端的Pod资源，这些Pod资源通常由第5章中介绍的各类控制器对象所创建和管理，因此Service资源通常要与控制器资源（最为常用的控制器之一是Deployment）协同使用以完成应用的创建和对外发布。

6.2.1 创建Service资源

创建Service对象的常用方法有两种：一是直接使用“`kubectl expose`”命令，这在前面第3章中已经介绍过其使用方式；另一个是使用资源配置文件，它与此前使用资源清单文件配置其他资源的方法类似。定义Service资源对象时，`spec`的两个较为常用的内嵌字段分别为`selector`和`ports`，分别用于定义使用的标签选择器和要暴露的端口。下面的配置清单是一个Service资源示例：

```
kind: Service
apiVersion: v1
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Service资源`myapp-svc`通过标签选择器关联至标签为“`app=myapp`”的各Pod对象，它会自动创建名为`myapp-svc`的Endpoints资源对象，并自动配置一个ClusterIP，暴露的端口由`port`字段进行指定，后端各Pod对象的端口则由`targetPort`给出，也可以使用同`port`字段的默认值。`myapp-svc`创建完成后，使用下面的命令即能获取相关的信息输出以了解资源的状态：

```
~]$ kubectl get svc myapp-svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp-svc	ClusterIP	10.107.208.93	<none>	80/TCP	56s

上面命令中的结果显示，`myapp-svc`的类型为默认的ClusterIP，其使用的地址自动配置为`10.107.208.93`。此类型的Service对象仅能通过此IP地址接受来自于集群内的客户端Pod中的请求。若集群上存在标签为“`app=myapp`”的Pod资源，则它们会被关联和创建，作为此Service对象的后端Endpoint对象，并负责接收相应的请求流量。类似下面的命

令可用于获取Endpoint资源的端点列表，其相关的端点是由第5章中的Deployment控制器创建的Pod对象的套接字信息组成的：

```
~]$ kubectl get endpoints myapp-svc
```

NAME	ENDPOINTS	AGE
myapp-svc	10.244.1.109:80,10.244.2.249:80,10.244.3.93:80	2m



提示 也可以不为Service资源指定.spec.selector属性值，其关联的Pod资源可由用户手动创建Endpoints资源进行定义。

Service对象创建完成后即可作为服务被各客户端访问，但要真正响应这些请求，还是要依赖于各后端的资源对象。

6.2.2 向Service对象请求服务

Service资源的默认类型为ClusterIP，它仅能接收来自于集群中的Pod对象中的客户端程序的访问请求。下面创建一个专用的Pod对象，利用其交互式接口完成访问测试。为了简单起见，这里选择直接创建一个临时使用的Pod对象作为交互式使用的客户端进行，它使用CirrOS镜像，默认的命令提示符为“/#”：

```
"/#":  
~]$ kubectl run cirros-$RANDOM --rm -it --image=cirros -- sh  
/ #
```



提示 CirrOS是设计用来进行云计算环境测试的Linux微型发行版，它拥有HTTP客户端工具curl等。

而后，在容器的交互式接口中使用crul命令对myapp-svc服务的ClusterIP（10.107.208.93）和Port（80/tcp）发起访问请求测试：

```
/ # curl http://10.107.208.93:80/  
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

myapp容器中的“/hostname.html”页面能够输出当前容器的主机名，可反复向myapp-svc的此URL路径发起多次请求以验证其调度的效果：

```
/ # for loop in 1 2 3 4; do curl http://10.107.208.93:80/hostname.html; done  
deploy-myapp-86b4b8c75d-rbhfl  
deploy-myapp-86b4b8c75d-mhbkd  
deploy-myapp-86b4b8c75d-xk8qb
```

当前Kubernetes集群的Service代理模式为iptables，它默认使用随机调度算法，因此Service会将客户端请求随机调度至与其关联的某个后端Pod资源上。命令取样次数越大，其调度效果也越接近于算法的目标效果。

6.2.3 Service会话粘性

Service资源还支持Session affinity（粘性会话或会话粘性）机制，它能够将来自同一个客户端的请求始终转发至同一个后端的Pod对象，这意味着它会影响调度算法的流量分发功用，进而降低其负载均衡的效果。因此，当客户端访问Pod中的应用程序时，如果有基于客户端身份保存某些私有信息，并基于这些私有信息追踪用户的活动等一类的需求时，那么应该启用session affinity机制。

Session affinity的效果仅会在一定时间期限内生效，默认值为10800秒，超出此时长之后，客户端的再次访问会被调度算法重新调度。另外，Service资源的Session affinity机制仅能基于客户端IP地址识别客户端身份，它会把经由同一个NAT服务器进行源地址转换的所有客户端识别为同一个客户端，调度粒度粗糙且效果不佳，因此，实践中并不推荐使用此种方法实现粘性会话。此节仅用于为读者介绍其功能及实现。

Service资源通过.spec.sessionAffinity和.spec.sessionAffinityConfig两个字段配置粘性会话。spec.sessionAffinity字段用于定义要使用的粘性会话的类型，它仅支持使用“None”和“ClientIP”两种属性值。

- None：不使用sessionAffinity，默认值。

- ClientIP：基于客户端IP地址识别客户端身份，把来自同一个源IP地址的请求始终调度至同一个Pod对象。

在启用粘性会话机制时，.spec.sessionAffinityConfig用于配置其会话保持的时长，它是一个嵌套字段，使用格式如下所示，其可用的时长范围为“1~86400”，默认为10800秒：

```
spec:
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    clientIP:
      timeoutSeconds: <integer>
```

例如，基于默认的10800秒的超时时长，使用下面的命令修改此前的myapp-svc使用Session affinity机制：

```
~]$ kubectl patch services myapp-svc -p '{"spec": {"sessionAffinity":  
"ClientIP"}}'  
service "myapp-svc" patched
```

而后再次于交互式客户端内测试其访问效果即可验证其会话粘性效果。

```
/ # for loop in 1 2 3 4; do curl http://10.107.208.93:80/hostname.html; done  
deploy-myapp-86b4b8c75d-rbhf1  
deploy-myapp-86b4b8c75d-rbhf1  
deploy-myapp-86b4b8c75d-rbhf1
```

测试完成后，为了保证本章后续的其他使用效果测试不受其影响，建议将其关闭。当然，用户也可以使用“kubectl edit”命令直接编辑活动Service对象的配置清单。

6.3 服务发现

微服务意味着存在更多的独立服务，但它们并非独立的个体，而是存在着复杂的依赖关系且彼此之间通常需要进行非常频繁地交互和通信的群体。然而，建立通信之前，服务和Service之间该如何获知彼此的地址呢？在Kubernetes系统上，Service为Pod中的服务类应用提供了一个稳定的访问入口，但Pod客户端中的应用如何得知某个特定Service资源的IP和端口呢？这个时候就需要引入服务发现（Service Discovery）的机制。

6.3.1 服务发现概述

简单来说，服务发现就是服务或者应用之间互相定位的过程。不过，服务发现并非新概念，传统的单体应用架构时代也会用到，只不过单体应用的动态性不强，更新和重新发布的频度较低，通常以月甚至以年计，基本上不会进行自动伸缩，因此服务发现的概念无须显性强调。在传统的单体应用网络位置发生变化时，由IT运维人员手工更新一下相关的配置文件基本上就能解决问题。但在微服务应用场景中，应用被拆分成众多的小服务，它们按需创建且变动频繁，配置信息基本无法事先写入配置文件中并及时跟踪和反映动态变化，因此服务发现的重要性便随之凸显。

服务发现机制的基本实现，一般是事先部署好一个网络位置较为稳定的服务注册中心（也称为服务总线），服务提供者（服务端）向注册中心注册自己的位置信息，并在变动后及时予以更新，相应地，服务消费者则周期性地从注册中心获取服务提供者的最新位置信息从而“发现”要访问的目标服务资源。复杂的服务发现机制还能够让服务提供者提供其描述信息、状态信息及资源使用信息等，以供消费者实现更为复杂的服务选择逻辑。

实践中，根据服务发现过程的实现方式，服务发现还可分为两种类型：客户端发现和服务端发现。

- 客户端发现：由客户端到服务注册中心发现其依赖到的服务的相关信息，因此，它需要内置特定的服务发现程序和发现逻辑。

- 服务端发现：这种方式需要额外用到一个称为中央路由器或服务均衡器的组件；服务消费者将请求发往中央路由器或者负载均衡器，由它们负责查询服务注册中心获取服务提供者的位置信息，并将服务消费者的请求转发给服务提供者。

由此可见，服务注册中心是服务发现得以落地的核心组件。事实上，DNS可以算是最为原始的服务发现系统之一，不过，在服务的动态性很强的场景中，DNS记录的传播速度可能会跟不上服务的变更速度，因此它并不适用于微服务环境。另外，传统实践中，常见的服务注册中心是ZooKeeper和etcd等分布式键值存储系统，不过，它们只能

提供基本的数据存储功能，距离实现完整的服务发现机制还有大量的二次开发任务需要完成。另外，它们更注重数据的一致性，这与有着更高的服务可用性要求的微服务发现场景中的需求不太相符。

Netflix的Eureka是目前较为流行的服务发现系统之一，它是专门开发用来实现服务发现的系统，以可用性目的为先，可以在多种故障期间保持服务发现和服务注册的功能可用，其设计原则遵从“存在少量的错误数据，总比完全不可用要好”。另一个同级别的实现是Consul，它是由HashiCorp公司提供的商业产品，不过该公司还提供了一个开源基础版本。它于服务发现的基础功能之外还提供了多数据中心的部署能力等一众出色的特性。

尽管传统的DNS系统不适于微服务环境中的服务发现，但SkyDNS项目（后来称kubedns）却是一个有趣的实现，它结合了古老的DNS技术和时髦的Go语言、Raft算法，并构建于etcd存储系统之上，为Kubernetes系统实现了一种服务发现机制。Service资源为Kubernetes提供了一个较为稳定的抽象层，这有点类似于服务端发现的方式，于是也就不存在DNS服务的时间窗口的问题。

Kubernetes自1.3版本开始，其用于服务发现的DNS更新为了kubeDNS，而类似的另一个基于较新的DNS的服务发现项目是由CNCF（Cloud Native Computing Foundation）孵化的CoreDNS，它基于Go语言开发，通过串接一组实现DNS功能的插件的插件链进行工作。自Kubernetes 1.11版本起，CoreDNS取代kubeDNS成为默认的DNS附件。不过，Kubernetes依然支持使用环境变量进行服务发现。

6.3.2 服务发现方式：环境变量

创建Pod资源时，`kubelet`会将其所属名称空间内的每个活动的Service对象以一系列环境变量的形式注入其中。它支持使用Kubernetes Service环境变量以及与Docker的links兼容的变量。

(1) Kubernetes Service环境变量

Kubernetes为每个Service资源生成包括以下形式的环境变量在内的一系列环境变量，在同一名称空间中创建的Pod对象都会自动拥有这些变量。

·{SVCNAME}_SERVICE_HOST

·{SVCNAME}_SERVICE_PORT



注意 如果SVCNAME中使用了连接线，那么Kubernetes会在定义为环境变量时将其转换为下划线。

(2) Docker Link形式的环境变量

Docker使用`--link`选项实现容器连接时所设置的环境变量形式，具体使用方式请参考Docker的相关文档。在创建Pod对象时，Kubernetes也会将与此形式兼容的一系列环境变量注入Pod对象中。

例如，在Service资源`myapp-svc`创建后创建的Pod对象中查看可用的环境变量，其中以`MYAPP_SVC_SERVICE`开头的表示Kubernetes Service环境变量，名称中不包含“SERVICE”字符串的环境变量为Docker Link形式的环境变量：

```
/ # printenv | grep MYAPP
MYAPP_SVC_PORT_80_TCP_ADDR=10.107.208.93
MYAPP_SVC_PORT_80_TCP_PORT=80
MYAPP_SVC_PORT_80_TCP_PROTO=tcp
MYAPP_SVC_PORT_80_TCP=tcp://10.107.208.93:80
MYAPP_SVC_SERVICE_HOST=10.107.208.93
MYAPP_SVC_SERVICE_PORT=80
MYAPP_SVC_PORT=tcp://10.107.208.93:80
```

基于环境变量的服务发现其功能简单、易用，但存在一定的局限，例如，仅有那些与创建的Pod对象在同一名称空间中且事先存在的Service对象的信息才会以环境变量的形式注入，那些处于非同一名称空间，或者是在Pod资源创建之后才创建的Service对象的相关环境变量则不会被添加。幸而，基于DNS的发现机制并不存在此类限制。

6.3.3 ClusterDNS和服务发现

Kubernetes系统之上用于名称解析和服务发现的ClusterDNS是集群的核心附件之一，集群中创建的每个Service对象，都会由其自动生成相关的资源记录。默认情况下，集群内各Pod资源会自动配置其作为名称解析服务器，并在其DNS搜索列表中包含它所属名称空间的域名后缀。

无论是使用kubeDNS还是CoreDNS，它们提供的基于DNS的服务发现解决方案都会负责解析以下资源记录（Resource Record）类型以实现服务发现。

（1）拥有ClusterIP的Service资源，需要具有以下类型的资源记录。

·A记录: <service>.<ns>.svc.<zone>. <ttl> IN A <cluster-ip>

·SRV记录: _<port>._<proto>.<service>.<ns>.svc.<zone>. <ttl> IN SRV <weight> <priority> <port-number> <service>.<ns>.svc.<zone>

·PTR记录: <d>.<c>..<a>.in-addr.arpa. <ttl> IN PTR <service>.<ns>.svc.<zone>

（2）Headless类型的Service资源，需要具有以下类型的资源记录。

·A记录: <service>.<ns>.svc.<zone>. <ttl> IN A <endpoint-ip>

·SRV记录: _<port>._<proto>.<service>.<ns>.svc.<zone>. <ttl> IN SRV <weight> <priority> <port-number> <hostname>.<service>.<ns>.svc.<zone>

·PTR记录: <d>.<c>..<a>.in-addr.arpa. <ttl> IN PTR <hostname>.<service>.<ns>.svc.<zone>

(3) ExternalName类型的Service资源，需要具有CNAME类型的资源记录。

·CNAME记录: <service>.<ns>.svc.<zone>. <ttl> IN CNAME
<extname>

名称解析和服务发现是Kubernetes系统许多功能得以实现的基础服务，它通常是集群安装完成后应该立即部署的附加组件。使用kubeadm初始化一个集群时，它甚至会自动进行部署。

6.3.4 服务发现方式：DNS

创建Service资源对象时，ClusterDNS会为它自动创建资源记录用于名称解析和服务注册，于是，Pod资源可直接使用标准的DNS名称来访问这些Service资源。每个Service对象相关的DNS记录包含如下两个。

·{SVCNAME}.{NAMESPACE}.{CLUSTER_DOMAIN}

·{SVCNAME}.{NAMESPACE}.svc.{CLUSTER_DOMAIN}

另外，在前面第2章的部署参数中，“--cluster-dns”指定了集群DNS服务的工作地址，“--cluster-domain”定义了集群使用的本地域名，因此，系统初始化时默认会将“cluster.local.”和主机所在的域“ilinux.io.”作为DNS的本地域使用，这些信息会在Pod创建时以DNS配置的相关信息注入它的/etc/resolv.conf配置文件中。例如，在此前创建的用于交互式Pod资源的客户端中查看其配置，命令如下：

```
/ # cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local ilinux.io
```

上述search参数中指定的DNS各搜索域，是以次序指定的几个域名后缀，具体如下所示。

·{NAMESPACE}.svc.{CLUSTER_DOMAIN}：如default.svc.cluster.local。

·svc.{CLUSTER_DOMAIN}：如svc.cluster.local。

·{CLUSTER_DOMAIN}：如cluster.local。

·{WORK_NODE_DOMAIN}：如ilinux.io。

例如，在此前创建的用于交互式Pod客户端中尝试请求解析myapp-svc的相关DNS记录：

```
/ # nslookup myapp-svc.default
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:       myapp-svc
Address 1: 10.107.208.93 myapp-svc.default.svc.cluster.local
```

解析时，“myapp-svc”服务名称的搜索次序依次是default.svc.cluster.local、svc.cluster.local、cluster.local和ilinux.io，因此基于DNS的服务发现不受Service资源所在的名称空间和创建时间的限制。上面的解析结果也正是默认的default名称空间中创建的myapp-svc服务的IP地址。

6.4 服务暴露

Service的IP地址仅在集群内可达，然而，总会有些服务需要暴露到外部网络中接受各类客户端的访问，例如分层架构应用中的前端**Web**应用程序等。此时，就需要在集群的边缘为其添加一层转发机制，以实现将外部请求流量接入到集群的**Service**资源之上，这种操作也称为发布服务到外部网络中。

6.4.1 Service类型

Kubernetes的Service共有四种类型：ClusterIP、NodePort、LoadBalancer和ExternalName。

- ClusterIP**：通过集群内部IP地址暴露服务，此地址仅在集群内部可达，而无法被集群外部的客户端访问，如图6-8所示。此为默认的Service类型。

- NodePort**：这种类型建立在ClusterIP类型之上，其在每个节点的IP地址的某静态端口（NodePort）暴露服务，因此，它依然会为Service分配集群IP地址，并将此作为NodePort的路由目标。简单来说，NodePort类型就是在工作节点的IP地址上选择一个端口用于将集群外部的用户请求转发至目标Service的ClusterIP和Port，因此，这种类型的Service既可如ClusterIP一样受到集群内部客户端Pod的访问，也会受到集群外部客户端通过套接字<NodeIP>: <NodePort>进行的请求。

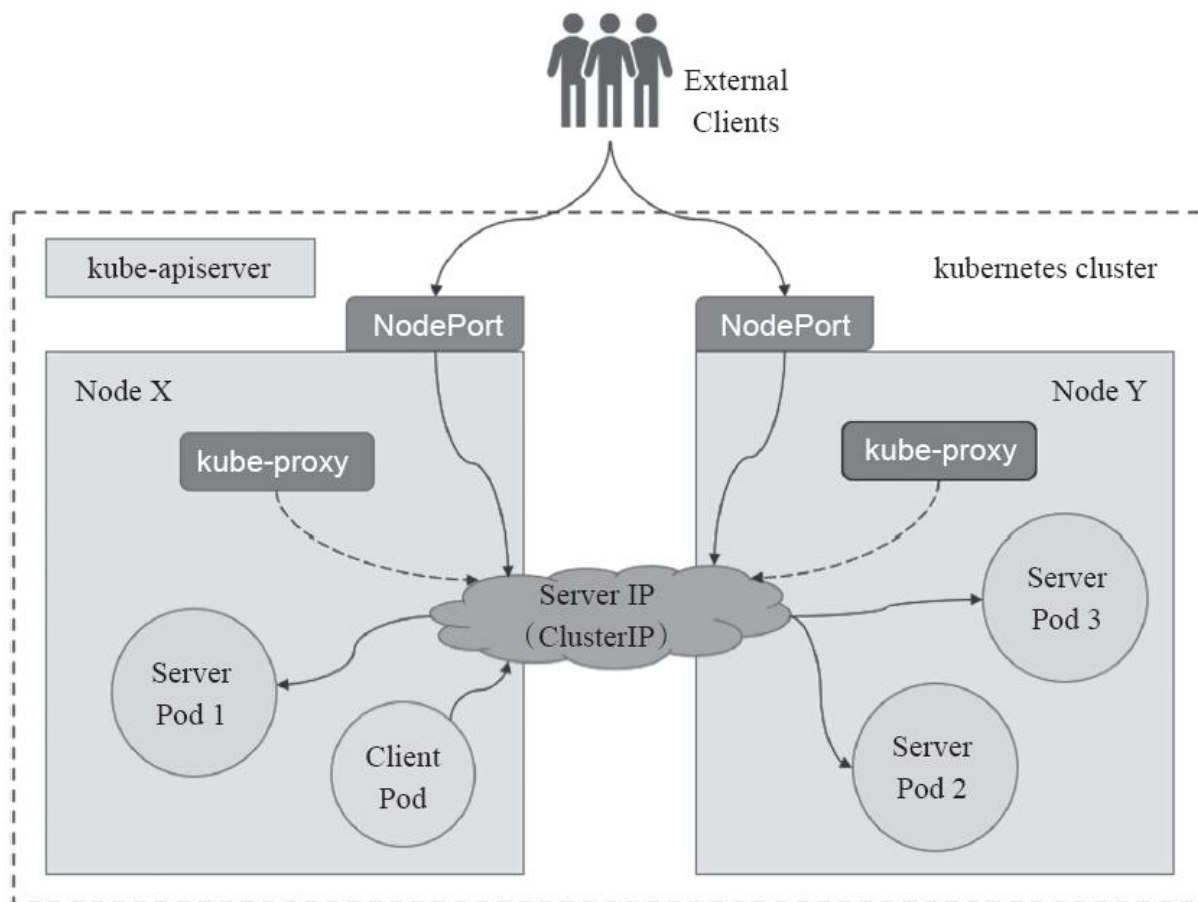


图6-8 NodePort Service类型

·**LoadBalancer**: 这种类型建构在NodePort类型之上，其通过cloud provider提供的负载均衡器将服务暴露到集群外部，因此LoadBalancer一样具有NodePort和ClusterIP。简而言之，一个LoadBalancer类型的Service会指向关联至Kubernetes集群外部的、切实存在的某个负载均衡设备，该设备通过工作节点之上的NodePort向集群内部发送请求流量，如图6-9所示。例如Amazon云计算环境中的ELB实例即为此类的负载均衡设备。此类型的优势在于，它能够把来自于集群外部客户端的请求调度至所有节点（或部分节点）的NodePort之上，而不是依赖于客户端自行决定连接至哪个节点，从而避免了因客户端指定的节点故障而导致的服务不可用。

·**ExternalName**: 其通过将Service映射至由externalName字段的内容指定的主机名来暴露服务，此主机名需要被DNS服务解析至CNAME类型的记录。换言之，此种类型并非定义由Kubernetes集群提供的服务，而是把集群外部的某服务以DNS CNAME记录的方式映射到集群内，从

而让集群内的Pod资源能够访问外部的Service的一种实现方式，如图6-10所示。因此，这种类型的Service没有ClusterIP和NodePort，也没有标签选择器用于选择Pod资源，因此也不会有Endpoints存在。

前面章节中创建的myapp-svc即为默认的ClusterIP类型Service资源，它仅能接收来自于集群中的Pod对象中的客户端程序的访问请求。如若需要将Service资源发布至网络外部，应该将其配置为NodePort或LoadBalancer类型，而若要把外部的服务发布于集群内容供Pod对象使用，则需要定义一个ExternalName类型的Service资源。如若使用kubedns，那么这种类型的实现将依赖于1.7及其以上版本的Kubernetes版本。

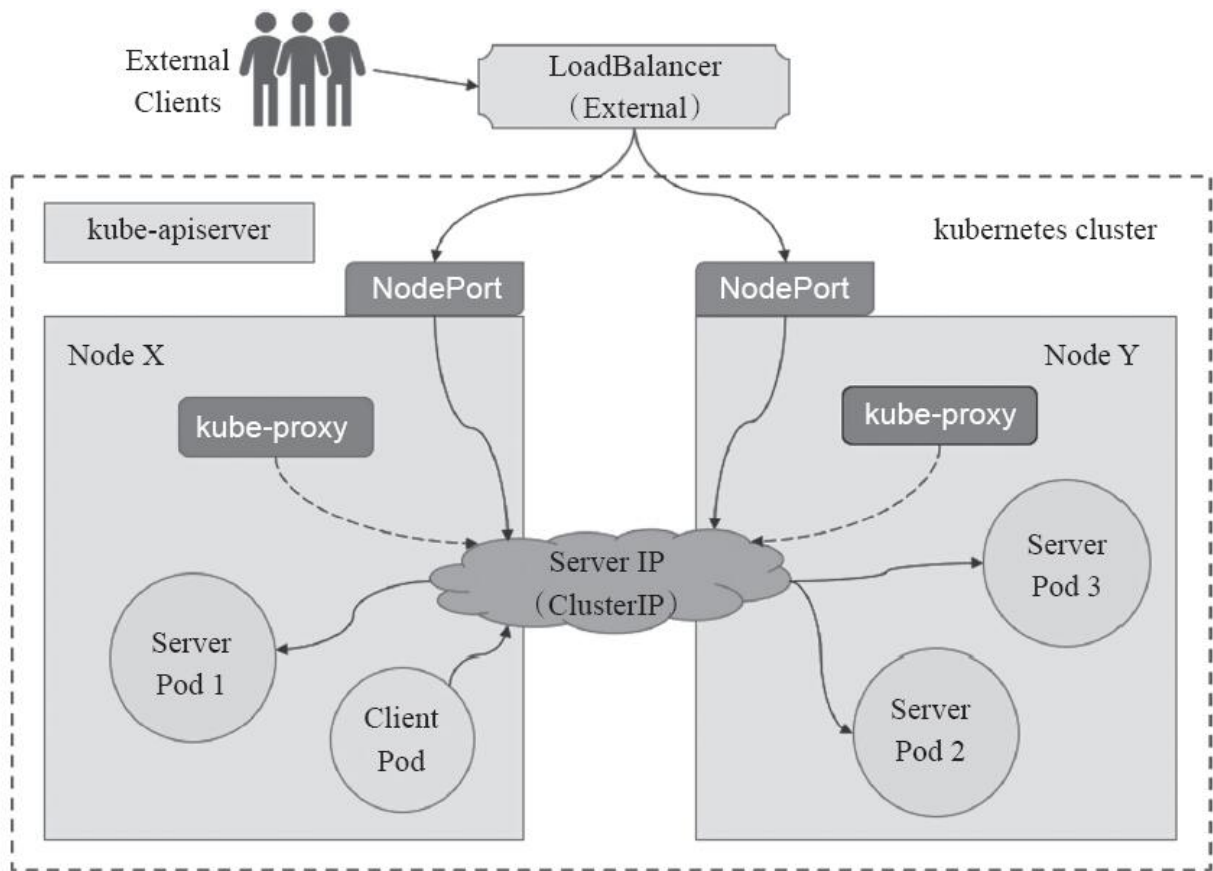


图6-9 LoadBalancer类型的Service

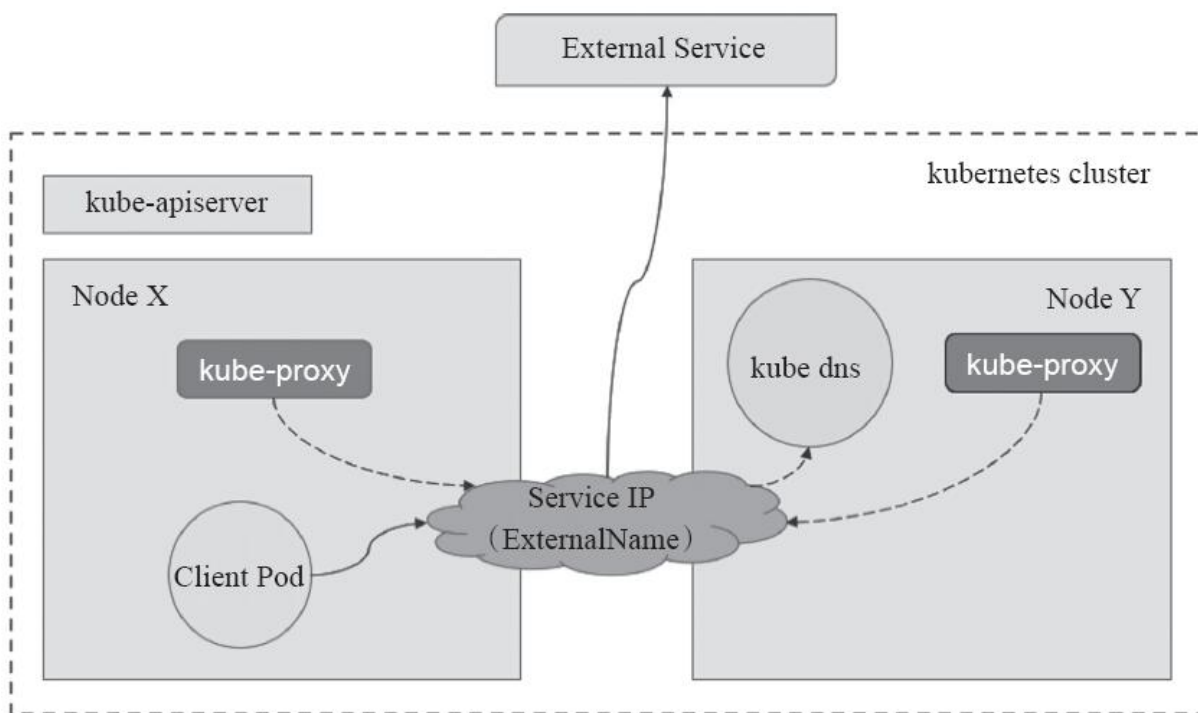


图6-10 ExternalName类型的Service

6.4.2 NodePort类型的Service资源

NodePort即节点Port，通常在安装部署Kubernetes集群系统时会预留一个端口范围用于NodePort，默认为30000~32767之间的端口。与ClusterIP类型的可省略.spec.type属性所不同的是，定义NodePort类型的Service资源时，需要通过此属性明确指定其类型名称。例如，下面配置清单中定义的Service资源对象myapp-svc-nodeport，它使用了NodePort类型，且人为指定其节点端口为32223：

```
kind: Service
apiVersion: v1
metadata:
  name: myapp-svc-nodeport
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 32223
```

实践中，并不鼓励用户自定义使用的节点端口，除非事先能够明确知道它不会与某个现存的Service资源产生冲突。无论如何，只要没有特别需求，留给系统自动配置总是较好的选择。使用创建命令创建上面的Service对象后即可了解其运行状态：

```
~]$ kubectl get services myapp-svc-nodeport
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp-svc-nodeport	NodePort	10.109.234.108	<none>	80:32223/TCP	3s

命令结果显示，NodePort类型的Service资源依然会被配置ClusterIP，事实上，它会作为节点从NodePort接入流量后转发的目标地址，目标端口则是与Service资源对应的spec.ports.port属性中定义的端口，如图6-11所示。

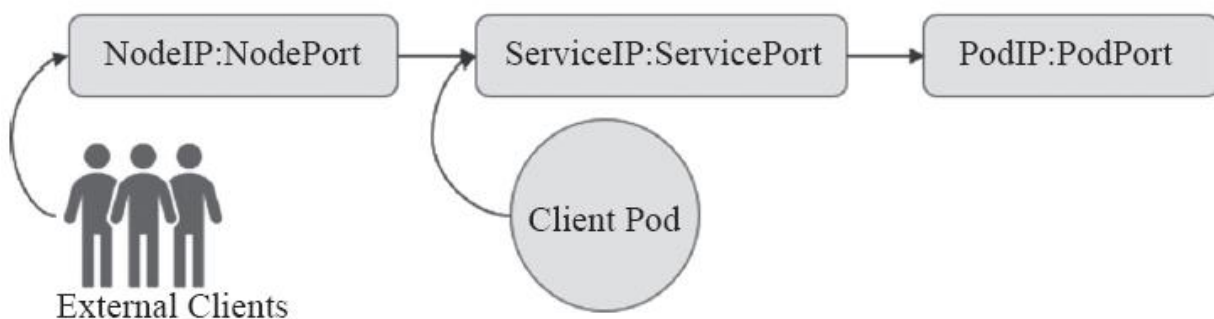


图6-11 请求流量转发过程

因此，对于集群外部的客户端来说，它们可经由任何一个节点的节点IP及端口访问NodePort类型的Service资源，而对于集群内的Pod客户端来说，依然可以通过ClusterIP对其进行访问。

6.4.3 LoadBalancer类型的Service资源

NodePort类型的Service资源虽然能够于集群外部访问得到，但外部客户端必须得事先得知NodePort和集群中至少一个节点的IP地址，且选定的节点发生故障时，客户端还得自行选择请求访问其他的节点。另外，集群节点很可能是某IaaS云环境中使用私有IP地址的VM，或者是IDC中使用私有地址的物理机，这类地址对互联网客户端不可达，因此，一般还应该在集群之外创建一个具有公网IP地址的负载均衡器，由它接入外部客户端的请求并调度至集群节点相应的NodePort之上。

IaaS云计算环境通常提供了LBaaS（Load Balancer as a Service）服务，它允许租户动态地在自己的网络中创建一个负载均衡设备。那些部署于此类环境之上的Kubernetes集群在创建Service资源时可以直接调用此接口按需创建一个软负载均衡器，而具有这种功能的Service资源即为LoadBalancer类型。不过，如果Kubernetes部署于裸的物理服务器之上，系统管理员也可以自行手动部署一个负载均衡器（推荐使用冗余配置），并配置其将请求流量调度至各节点的NodePort之上即可。

下面是一个LoadBalancer类型的Service资源配置清单，若Kubernetes系统满足其使用条件，即可自行进行应用测试。需要注意的是，有些环境中可能还需要为Service资源的配置定义添加Annotations，必要时请自行参考Kubernetes文档中的说明：

```
kind: Service
apiVersion: v1
metadata:
  name: myapp-svc-lb
spec:
  type: LoadBalancer
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32223
```

进一步地，在IaaS环境支持手动指定IP地址时，用户还可以使用`.spec.loadBalancerIP`指定创建的负载均衡器使用的IP地址，并可使用`.spec.loadBalancerSourceRanges`指定负载均衡器允许的客户端来源的地址范围。

6.4.4 ExternalName Service

ExternalName类型的**Service**资源用于将集群外部的服务发布到集群中以供**Pod**中的应用程序访问，因此，它不需要使用标签选择器关联任何的**Pod**对象，但必须要使用**spec.externalName**属性定义一个**CNAME**记录用于返回外部真正提供服务的主机的别名，而后通过**CNAME**记录值获取到相关主机的IP地址。

下面是一个**ExternalName**类型的**Service**资源示例，名为**external-redis-svc**，相应的**externalName**为“**redis.ilinux.io**”：

```
kind: Service
apiVersion: v1
metadata:
  name: external-redis-svc
  namespace: default
spec:
  type: ExternalName
  externalName: redis.ilinux.io
  ports:
  - protocol: TCP
    port: 6379
    targetPort: 6379
    nodePort: 0
  selector: {}
```

待**Service**资源**external-redis-svc**创建完成后，各**Pod**对象即可通过**external-redis-svc**或其**FQDN**格式的名称**external-redis-svc.default.svc.cluster.local**访问相应的服务。ClusterDNS会将此名称以**CNAME**格式解析为**spec.externalName**字段中的名称，而后通过**DNS**服务将其解析为相应的主机的IP地址。例如，通过此前创建的交互**Pod**资源客户端进行服务名称解析：

```
/ # nslookup external-redis-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:        external-redis-svc
Address 1: 45.54.44.100 100.44.54.45.ptr.anycast.net
```

由于ExternalName类型的Service资源实现于DNS级别，客户端将直接接入外部的服务而完全不需要服务代理，因此，它也无须配置ClusterIP，此种类型的服务也称为Headless Service。

6.5 Headless类型的Service资源

Service对象隐藏了各Pod资源，并负责将客户端的请求流量调度至该组Pod对象之上。不过，偶尔也会存在这样一类需求：客户端需要直接访问Service资源后端的所有Pod资源，这时就应该向客户端暴露每个Pod资源的IP地址，而不再是中间层Service对象的ClusterIP，这种类型的Service资源便称为Headless Service。

Headless Service对象没有ClusterIP，于是kube-proxy便无须处理此类请求，也就更没有了负载均衡或代理它的需要。在前端应用拥有自有的其他服务发现机制时，Headless Service即可省去定义ClusterIP的需求。至于如何为此类Service资源配置IP地址，则取决于它的标签选择器的定义。

- 具有标签选择器：端点控制器（Endpoints Controller）会在API中为其创建Endpoints记录，并将ClusterDNS服务中的A记录直接解析到此Service后端的各Pod对象的IP地址上。

- 没有标签选择器：端点控制器（Endpoints Controller）不会在API中为其创建Endpoints记录，ClusterDNS的配置分为两种情形，对ExternalName类型的服务创建CNAME记录，对其他三种类型来说，为那些与当前Service共享名称的所有Endpoints对象创建一条记录。

6.5.1 创建Headless Service资源

配置Service资源配置清单时，只需要将ClusterIP字段的值设置为“None”即可将其定义为Headless类型。下面是一个Headless Service资源配置清单示例，它拥有标签选择器：

```
kind: Service
apiVersion: v1
metadata:
  name: myapp-headless-svc
spec:
  clusterIP: None
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 80
    name: httpport
```

使用资源创建命令“`kubectl create`”或“`kubectl apply`”完成资源创建后，使用相关的查看命令获取Service资源的相关信息便可以看出，它没有ClusterIP，不过，如果标签选择器能够匹配到相关的Pod资源，它便拥有Endpoints记录，这些Endpoints对象会作为DNS资源记录名称myapp-headless-svc查询时的A记录解析结果：

```
~]$ kubectl describe svc myapp-headless-svc
.....
Endpoints:          10.244.1.113:80,10.244.2.13:80,10.244.3.104:80
.....
```

6.5.2 Pod资源发现

根据Headless Service的工作特性可知，它记录于ClusterDNS的A记录的相关解析结果是后端Pod资源的IP地址，这就意味着客户端通过此Service资源的名称发现的是各Pod资源。下面依然选择创建一个专用的测试Pod对象，而后通过其交互式接口进行测试：

```
~]$ kubectl run cirros-$RANDOM --rm -it --image=cirros -- sh
/ #
/ # nslookup myapp-headless-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:        myapp-headless-svc
Address 1: 10.244.2.13
Address 2: 10.244.1.113
Address 3: 10.244.3.104
```

其解析结果正是Headless Service通过标签选择器关联到的所有Pod资源的IP地址。于是，客户端向此Service对象发起的请求将直接接入到Pod资源中的应用之上，而不再由Service资源进行代理转发，它每次接入的Pod资源则是由DNS服务器接收到查询请求时以轮询（roundrobin）的方式返回的IP地址。

6.6 Ingress资源

Kubernetes提供了两种内建的云端负载均衡机制（cloud load balancing）用于发布公共应用，一种是工作于传输层的Service资源，它实现的是“TCP负载均衡器”，另一种是Ingress资源，它实现的是“HTTP（S）负载均衡器”。

（1）TCP负载均衡器

无论是iptables还是ipvs模型的Service资源都配置于Linux内核中的Netfilter之上进行四层调度，是一种类型更为通用的调度器，支持调度HTTP、MySQL等应用层服务。不过，也正是由于工作于传输层从而使得它无法做到类似卸载HTTPS中的SSL会话等一类操作，也不支持基于URL的请求调度机制，而且，Kubernetes也不支持为此类负载均衡器配置任何类型的健康状态检查机制。

（2）HTTP（S）负载均衡器

HTTP（S）负载均衡器是应用层负载均衡机制的一种，支持根据环境做出更好的调度决策。与传输层调度器相比，它提供了诸如可自定义URL映射和TLS卸载等功能，并支持多种类型的后端服务器健康状态检查机制。

6.6.1 Ingress和Ingress Controller

Kubernetes中，Service资源和Pod资源的IP地址仅能用于集群网络内部的通信，所有的网络流量都无法穿透边界路由器（Edge Router）以实现集群内外通信。尽管可以为Service使用NodePort或LoadBalancer类型通过节点引入外部流量，但它依然是4层流量转发，可用的负载均衡器也为传输层负载均衡机制。

Ingress是Kubernetes API的标准资源类型之一，它其实就是一组基于DNS名称（host）或URL路径把请求转发至指定的Service资源的规则，用于将集群外部的请求流量转发至集群内部完成服务发布。然而，Ingress资源自身并不能进行“流量穿透”，它仅是一组路由规则的集合，这些规则要想真正发挥作用还需要其他功能的辅助，如监听某套接字，然后根据这些规则的匹配机制路由请求流量。这种能够为Ingress资源监听套接字并转发流量的组件称为Ingress控制器（Ingress Controller）。



注意 不同于Deployment控制器等，Ingress控制器并不直接运行为kube-controller-manager的一部分，它是Kubernetes集群的一个重要附件，类似于CoreDNS，需要在集群上单独部署。

Ingress控制器可以由任何具有反向代理（HTTP/HTTPS）功能的服务程序实现，如Nginx、Envoy、HAProxy、Vulcand和Traefik等。Ingress控制器自身也是运行于集群中的Pod资源对象，它与被代理的运行Pod资源的应用运行于同一网络中，如图6-12中ingress-nginx与pod1、pod3等的关系所示。

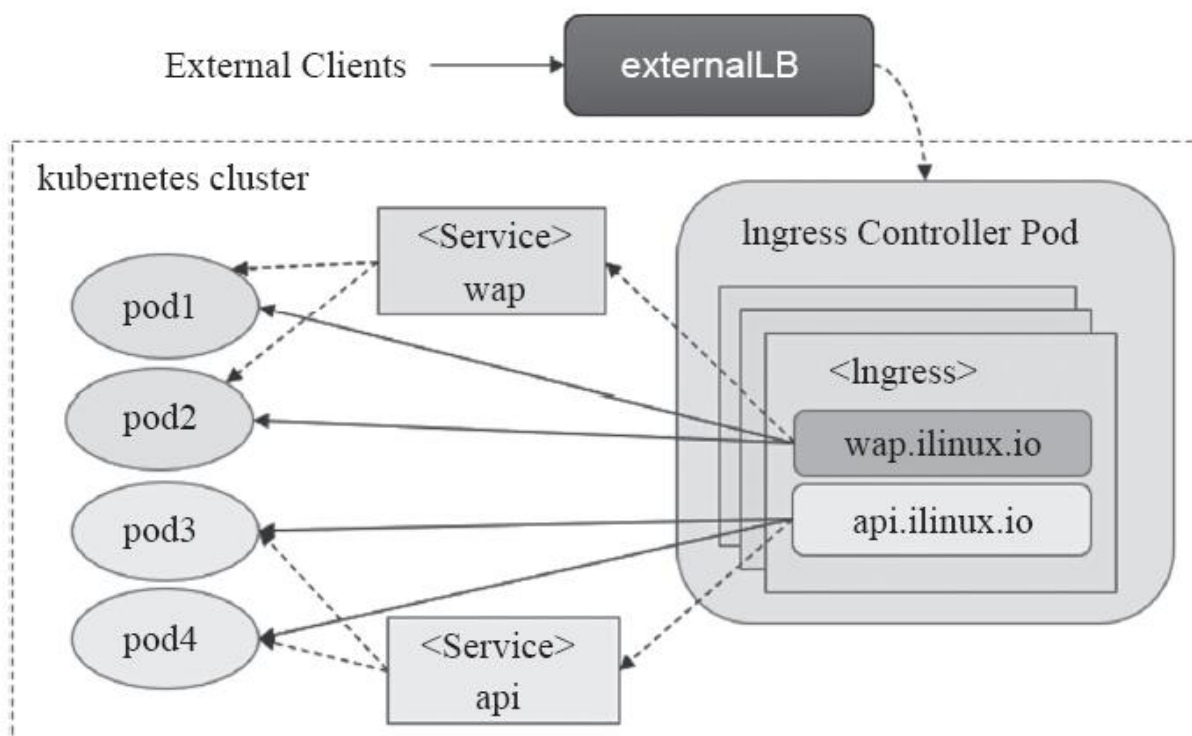


图6-12 Ingress与Ingress Controller

另一方面，使用Ingress资源进行流量分发时，Ingress控制器可基于某Ingress资源定义的规则将客户端的请求流量直接转发至与服务对应的后端Pod资源之上，这种转发机制会绕过Service资源，从而省去了由kube-proxy实现的端口代理开销。如图6-12所示，Ingress规则需要由一个Service资源对象辅助识别相关的所有Pod对象，但ingress-nginx控制器可经由api.ilinux.io规则的定义直接将请求流量调度至pod3或pod4，而无须经由Service对象API的再次转发，WAP相关规则的作用方式与此类同。

6.6.2 创建Ingress资源

Ingress资源是基于HTTP虚拟主机或URL的转发规则，它在资源配置清单的spec字段中嵌套了rules、backend和tls等字段进行定义。下面的示例中定义了一个Ingress资源，它包含了一个转发规则，把发往www.ilinux.io的请求代理给名为myapp-svc的Service资源：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: www.ilinux.io
    http:
      paths:
      - backend:
          serviceName: myapp-svc
          servicePort: 80
```

上面资源清单中的annotations用于识别其所属的Ingress控制器的类别，这一点在集群上部署有多个Ingress控制器时尤为重要。Ingress Spec中的字段是定义Ingress资源的核心组成部分，它主要嵌套如下三个字段。

·rules<Object>：用于定义当前Ingress资源的转发规则列表；未由rules定义规则，或者没有匹配到任何规则时，所有流量都会转发到由backend定义的默认后端。

·backend<Object>：默认的后端用于服务那些没有匹配到任何规则的请求；定义Ingress资源时，至少应该定义backend或rules两者之一；此字段用于让负载均衡器指定一个全局默认的后端。

·tls<Object>：TLS配置，目前仅支持通过默认端口443提供服务；如果要配置指定的列表成员指向了不同的主机，则必须通过SNI TLS扩展机制来支持此功能。

backend对象的定义由两个必选的内嵌字段组成：**serviceName**和**servicePort**，分别用于指定流量转发的后端目标**Service**资源的名称和端口。

rules对象由一系列配置**Ingress**资源的**host**规则组成，这些**host**规则用于将一个主机上的某个**URL**路径映射至相关的后端**Service**对象，它的定义格式如下：

```
spec:
  rules:
  - host: <String>
    http:
      paths:
        backend:
          serviceName: <String>
          servicePort: <String>
          path: <String>
```

注意，**.spec.rules.host**属性值目前不支持使用**IP**地址，也不支持后跟“: **PORT**”格式的端口号，且此字段值留空表示通配所有的主机名。

tls对象由两个内嵌字段组成，仅在定义**TLS**主机的转发规则时才需要定义此类对象。

- hosts**: 包含于使用的**TLS**证书之内的主机名称字符串列表，因此，此处使用的主机名必须匹配**tlsSecret**中的名称。

- secretName**: 用于引用**SSL**会话的**secret**对象名称，在基于**SNI**实现多主机路由的场景中，此字段为可选。

6.6.3 Ingress资源类型

基于HTTP暴露的每个Service资源均可发布于一个独立的FQDN主机名之上，如“www.ik8s.io”；也可发布于某主机的URL路径之上，从而将它们整合到同一个Web站点，如“www.ik8s.io/grafana”。至于是否需要发布为HTTPS类型的应用则取决于用户的业务需求。

1.单Service资源型Ingress

暴露单个服务的方法有很多种，如服务类型中的NodePort、LoadBalancer等，不过一样可以考虑使用Ingress来暴露服务，此时只需要为Ingress指定“default backend”即可。例如下面的示例：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  backend:
    serviceName: my-svc
    servicePort: 80
```

Ingress控制器会为其分配一个IP地址接入请求流量，并将它们转至示例中的my-svc后端。

2.基于URL路径进行流量分发

垂直拆分或微服务架构中，每个小的应用都有其专用的Service资源暴露服务，但在对外开放的站点上，它们可能是财经、新闻、电商、无线端或API接口等一类的独立应用，可通过主域名的URL路径（path）分别接入，例如，www.ilinux.io/api、www.ilinux.io/wap等，用于发布集群内名称为API和WAP的Services资源。于是，可对应地创建一个如下的Ingress资源，它将对www.ilinux.io/api的请求统统转发至API Service资源，将对www.ilinux.io/wap的请求转发至WAP Service资源：

```
apiVersion: extensions/v1beta1
kind: Ingress
```

```
metadata:
  name: test
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: www.ilinux.io
    http:
      paths:
      - path: /wap
        backend:
          serviceName: wap
          servicePort: 80
      - path: /api
        backend:
          serviceName: api
          servicePort: 80
```



注意 目前，ingress-nginx似乎尚且不能很好地支持基于annotations进行URL映射。这就意味着，在ingress-nginx上，此项功能尚且不能使用。具体信息请参考这个链接中的讨论，<https://github.com/istio/istio/issues/585>。

3. 基于主机名称的虚拟主机

上面类型2中描述的需求，也可以将每个应用分别以独立的FQDN主机名进行输出，如wap.ik8s.io和api.ik8s.io，这两个主机名解析到external LB（如图6-12所示）的IP地址之上，分别用于发布集群内部的WAP和API这两个Service资源。这种实现方案其实就是Web站点部署中的“基于主机名的虚拟主机”，将多个FQDN解析至同一个IP地址，然后根据“主机头”（Host header）进行转发。下面是以独立FQDN主机形式发布服务的Ingress资源示例：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: api.ik8s.io
    http:
      paths:
      - backend:
          serviceName: api
          servicePort: 80
  - host: wap.ik8s.io
    http:
      paths:
```

```
- backend:
  serviceName: wap
  servicePort: 80
```

4.TLS类型的Ingress资源

这种类型用于以HTTPS发布Service资源，基于一个含有私钥和证书的Secret对象（后面章节中会详细讲述）即可配置TLS协议的Ingress资源，目前来说，Ingress资源仅支持单TLS端口，并且还会卸载TLS会话。在Ingress资源中引用此Secret即可让Ingress控制器加载并配置为HTTPS服务。

下面是一个简单的TLS类型的Ingress资源示例：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
  - secretName: ikubernetesSecret
  backend:
    serviceName: homesite
    servicePort: 80
```

6.6.4 部署Ingress控制器（Nginx）

Ingress控制器自身是运行于Pod中的容器应用，一般是Nginx或Envoy一类的具有代理及负载均衡功能的守护进程，它监视着来自于API Server的Ingress对象状态，并以其规则生成相应的应用程序专有格式的配置文件并通过重载或重启守护进程而使新配置生效。例如，对于Nginx来说，Ingress规则需要转换为Nginx的配置信息。简单来说，Ingress控制器其实就是托管于Kubernetes系统之上的用于实现在应用层发布服务的Pod资源，它将跟踪Ingress资源并实时生成配置规则。那么，同样运行于Pod资源的Ingress控制器进程又该如何接入外部的请求流量呢？常用的解决方案有如下两种。

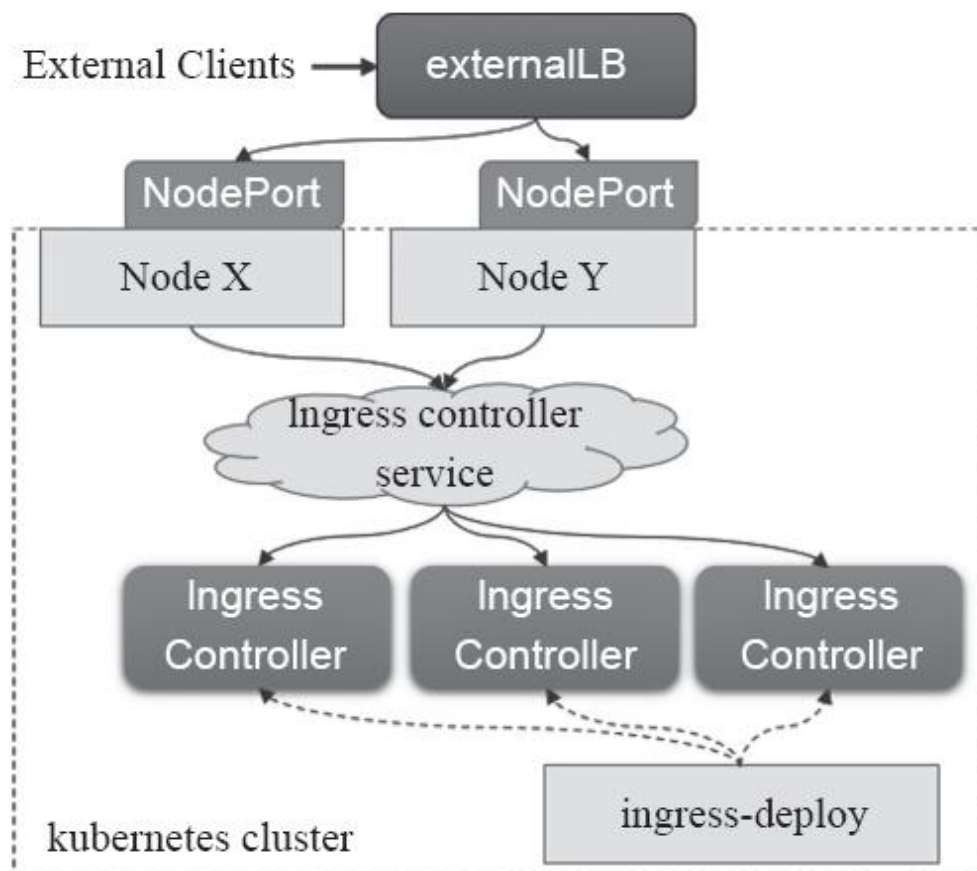


图6-13 使用专用的Service对象为Ingress控制器接入外部流量

·以Deployment控制器管理Ingress控制器的Pod资源，并通过NodePort或LoadBalancer类型的Service对象为其接入集群外部的请求流

量，这就意味着，定义一个Ingress控制器时，必须在其前端定义一个专用的Service资源，如图6-13所示。

- 借助于DaemonSet控制器，将Ingress控制器的Pod资源各自以单一实例的方式运行于集群的所有或部分工作节点之上，并配置这类Pod对象以hostPort（如图6-14a）或hostNetwork（如图6-14b）的方式在当前节点接入外部流量。

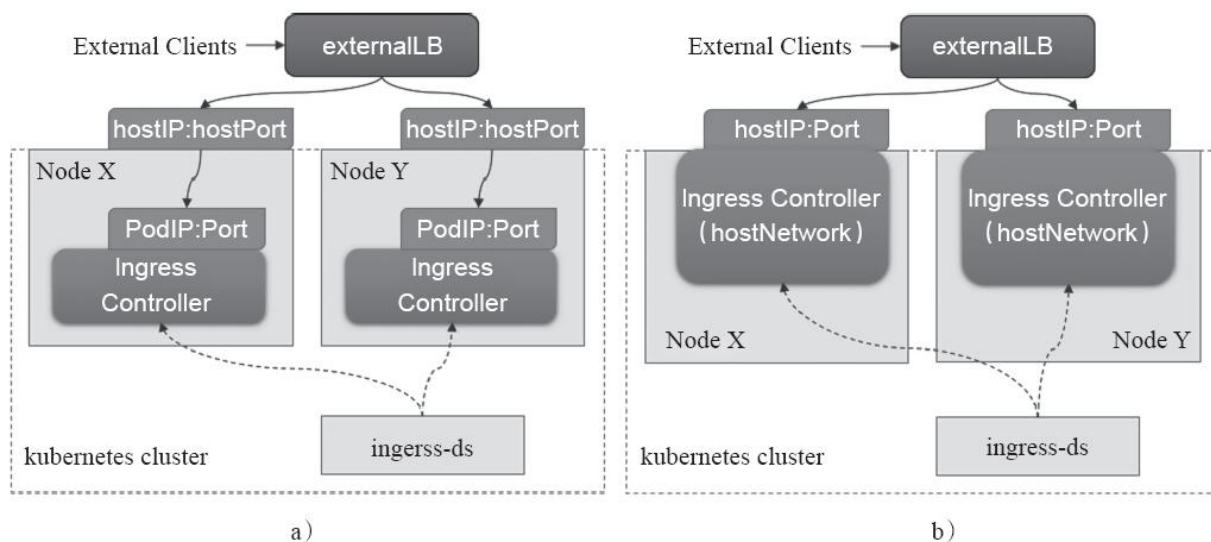


图6-14 以hostPort或hostNetwork的方式为Ingress控制器接入外部流量

以ingress-nginx项目为例，部署Ingress Nginx控制器的配置文件被切割存放在了多个不同的文件中，并集中存储于其源码deploy子目录下，同时，为了方便用户部署，它还将所需的资源全部集成为一个配置文件mandatory.yaml:

```
~]$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/mandatory.yaml
```

因为需要下载相关的镜像文件，因此前面部署过程中的Pod资源的创建需要等待一段时间才能完成，具体时长要取决于网络的可用状况。可使用如下命令持续监控创建过程，待其状态为“**Running**”之后即表示运行正常：

```
~]$ kubectl get pods -n ingress-nginx --watch
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

default-http-backend-6586bc58b6-cw7c6	1/1	Running	0	3m
nginx-ingress-controller-7675fd6cdb-kvsh2	1/1	Running	0	3m

在线的配置清单中采用了基于Deployment控制器部署Ingress Nginx的方式，因此接入外部流量之前还需要手动为其创建相关的NodePort或LoadBalancer类型的Service资源对象，下面的配置清单示例中对类型定义了NodePort，并明确指定了易记的端口和IP地址，以方便用户使用：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
spec:
  type: NodePort
  clusterIP: 10.99.99.99
  ports:
    - port: 80
      name: http
      nodePort: 30080
    - port: 443
      name: https
      nodePort: 30443
  selector:
    app.kubernetes.io/name: ingress-nginx
```

将上面的配置信息保存于文件中，如nginx-ingress-service.yaml，而后执行如下命令完成资源的创建。注意，其标签选择器应该与mandatory.yaml配置清单中的Deployment控制器nginx-ingress-controller的选择器保持一致：

```
~]# kubectl apply -f nginx-ingress-service.yaml
```



注意 如果读者的集群运行支持LBaaS的IaaS云环境，则可以将其类型指定为LoadBalancer，这样直接就有了可用的external-LB。

确认Service对象nginx-ingress-controller的状态没有问题后即可于集群外部对其发起访问测试，目标URL为<http://<NodeIP>:30080> 或 <http://<NodeIP>:30443>，确认可接收到响应报文后即表示Ingress Nginx部署完成。不过，本示例中尚且缺少一个可用的外部负载均衡器，如图6-13中所示的“external-LB”，因此，访问测试时暂时还只能使用<http://<NodeIp>:<NodePort>> 进行。

6.7 案例：使用Ingress发布tomcat

假设有这样一套环境：Kubernetes集群上的tomcat-deploy控制器生成了两个运行于Pod资源中的tomcat实例，tomcat-svc是将它们统一暴露于集群中的访问入口。现在需要通过Ingress资源将tomcat-svc发布给集群外部的客户端访问。具体的需求和规划如图6-15所示。

为了便于读者理解，下面的测试操作过程将把每一步分解开来放在单独的一节中进行。

6.7.1 准备名称空间

假设本示例中创建的所有资源都位于新建的`testing`名称空间中，与其他的资源在逻辑上进行隔离，以方便管理。下面的配置信息保存于`testing-namespace.yaml`资源清单文件中：

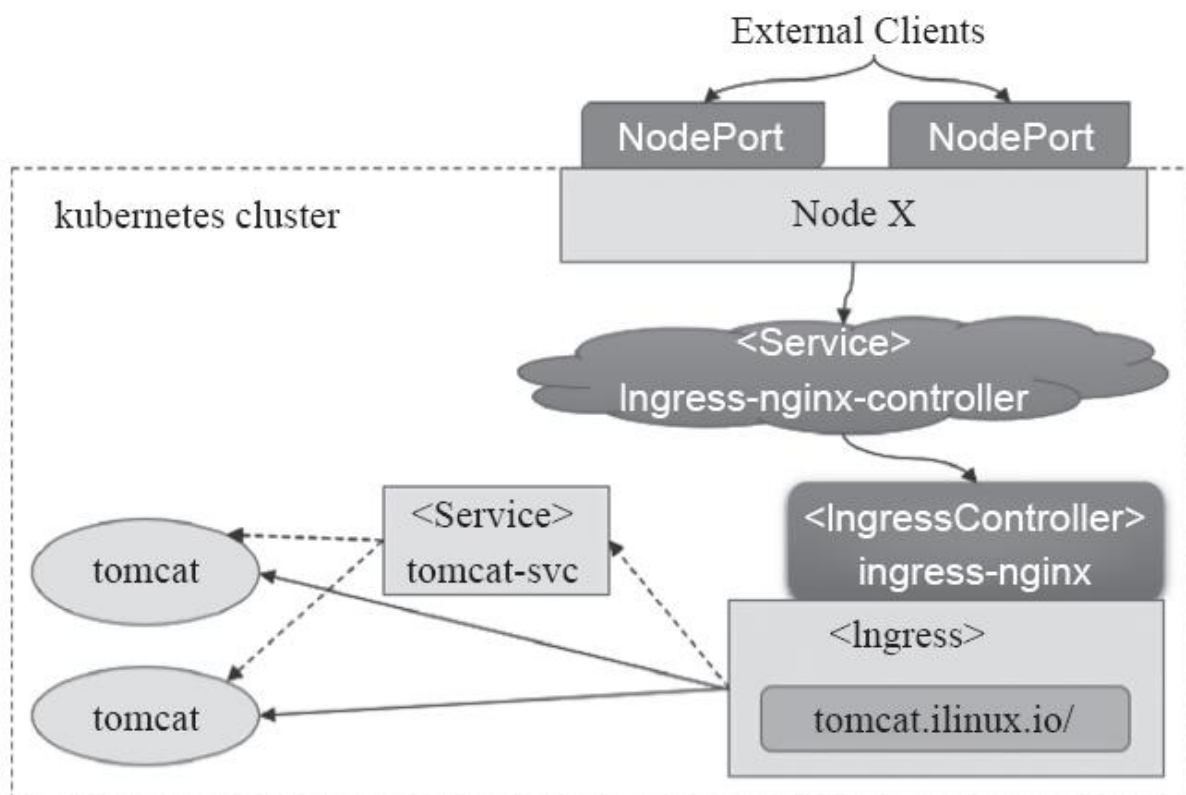


图6-15 Ingress发布应用示例拓扑图

```
kind: Namespace
apiVersion: v1
metadata:
  name: testing
  labels:
    env: testing
```

而后运行创建命令完成资源的创建，并确认资源的存在：

```
-]$ kubectl apply -f testing-namespace.yaml
namespace "testing" created
-]$ kubectl get namespaces testing
```

NAME	STATUS	AGE
testing	Active	8s

6.7.2 部署tomcat实例

在此示例中，**tomcat**应用本身代表着运行于**tomcat**容器中的一个实际应用。具体实践中，它通常应该是包含了某应用程序的**war**文件的镜像文件。下面的配置清单使用了**Deployment**控制器于**testing**中部署**tomcat**相关的**Pod**对象，它保存于**tomcat-deploy.yaml**文件中：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deploy
  namespace: testing
spec:
  replicas: 2
  selector:
    matchLabels:
      app: tomcat
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
        - name: tomcat
          image: tomcat:8.0.50-jre8-alpine
          ports:
            - containerPort: 8080
              name: httpport
            - containerPort: 8009
              name: ajpport
```

运行资源创建命令完成**Deployment**控制器和**Pod**资源的创建，命令如下：

```
~]$ kubectl apply -f tomcat-deploy.yaml
```

接着运行命令以确认其成功完成，且各**Pod**已经处于正常运行状态中：

```
~]$ kubectl get pods -n testing
```

NAME	READY	STATUS	RESTARTS	AGE
tomcat-deploy-6cf8468f7f-5d7tb	1/1	Running	0	1m
tomcat-deploy-6cf8468f7f-9fvxx	1/1	Running	0	1m

实践中，如果需要更多的Pod资源承载用户访问，那么使用Deployment控制器的规模伸缩机制即可完成，或者直接修改上面的配置文件并执行“`kubectl apply`”命令重新进行应用。

6.7.3 创建Service资源

Ingress资源仅通过Service资源识别相应的Pod资源，获取其IP和端口，而后Ingress控制器即可直接使用各Pod对象的IP地址与它直接进行通信，而不经由Service资源的代理和调度，因此Service资源的ClusterIP对Ingress控制器来说一无所用。不过，若集群内的其他Pod客户端需要与其通信，那么保留ClusterIP似乎也是很有必要的。

下面的配置文件中定义了Service资源tomcat-svc，它通过标签选择器将相关的Pod对象归于一组，并通过80/TCP端口暴露Pod对象的8080/TCP端口。如果需要暴露容器的8009/TCP端口，那么只需要将其以类似的格式配置于列表中即可：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-svc
  namespace: testing
  labels:
    app: tomcat-svc
spec:
  selector:
    app: tomcat
  ports:
  - name: http
    port: 80
    targetPort: 8080
    protocol: TCP
```

运行资源创建命令完成Service资源的创建：

```
~]$ kubectl apply -f tomcat-svc.yaml
```

接着运行命令以确认其成功完成：

```
~]$ kubectl get svc tomcat-svc -n testing
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
tomcat-svc	ClusterIP	10.108.72.237	<none>	80/TCP	8s

6.7.4 创建Ingress资源

通过Ingress资源的FQDN主机名或URL路径等类型发布的服务，只有用户的访问请求能够匹配到其.spec.rules.host字段定义的主机时才能被相应的规则处理。如果要明确匹配用户的处理请求，比如希望将那些发往tomcat.ilinux.io主机的所有请求代理至tomcat-svc资源的后端Pod，则可以使用如下命令配置文件中的内容：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: tomcat
  namespace: testing
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: tomcat.ilinux.io
    http:
      paths:
      - path:
          backend:
            serviceName: tomcat-svc
            servicePort: 80
```

运行资源创建命令完成Service资源的创建：

```
~]$ kubectl apply -f tomcat-ingress.yaml
```

而后通过详细信息确认其创建成功完成，并且已经正确关联到相应的tomcat-svc资源上：

```
~]$ kubectl describe ingresses -n testing
Name:                tomcat
Namespace:           testing
Address:
Default backend:     default-http-backend:80 (<none>)
Rules:
  Host                Path  Backends
  ----                -
  tomcat.ilinux.io    tomcat-svc:80 (<none>)

Annotations:
Events:
```


Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	CREATE	3s	nginx-ingress-controller	Ingress testing/tomcat

接下来即可通过Ingress控制器的前端Service资源的NodePort来访问此服务，在6.6.4节中，此Service资源的ClusterIP被明确定义为10.99.99.99，并以节点端口30080映射Ingress控制器的80端口。因此，这里使用Ingress中定义的主机名tomcat.ilinux.io: 30080即可访问tomcat应用，图6-16所示的是访问页面的效果。当然，实践中，其前端应该有一个外部的负载均衡设备接收并调度此类请求。

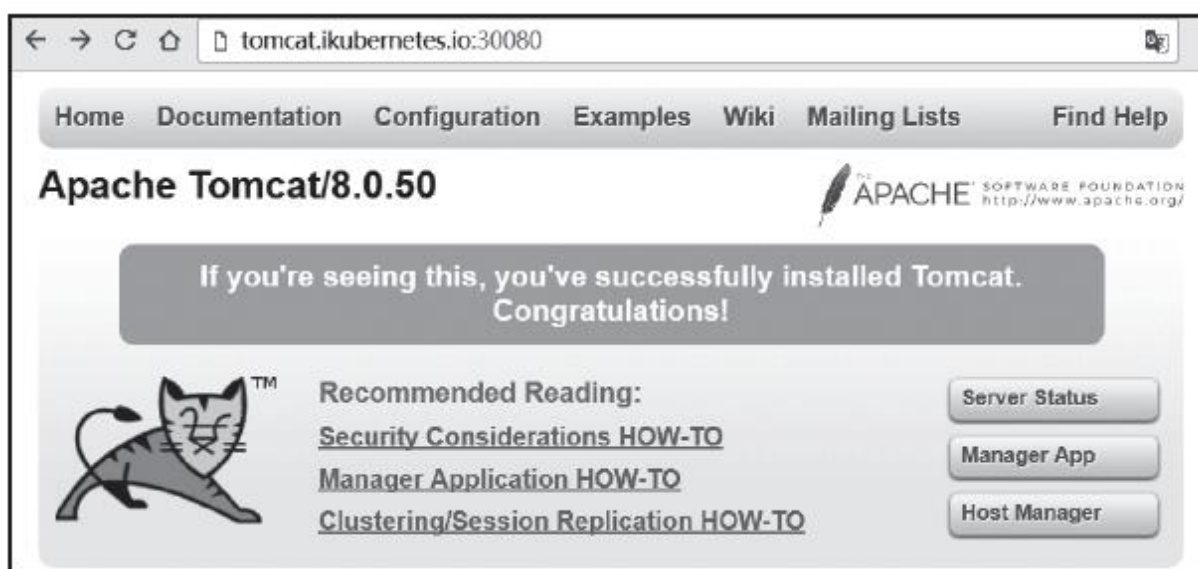


图6-16 访问Ingress资源代理的tomcat应用

不过，用户对tomcat.ilinux.io主机之外的地址发起的彼此Ingress规则匹配到的请求将发往Ingress控制器的默认的后端，即default-http-backend，它通常只能返回一个404提示信息。用户也可按需自定义默认后端，例如，如下面的配置文件片断所示，它通过spec.backend定义了所有无法由此Ingress匹配的访问请求都由相应的后端default-svc这个Service资源来处理：

```
spec:
  backend:
    serviceName: default-svc
    servicePort: 80
```

6.7.5 配置TLS Ingress资源

一般来说，如果有基于HTTPS通信的需求，那么它应该由外部的负载均衡器（external-LB）予以实现，并在SSL会话卸载后将访问请求转发到Ingress控制器。不过，如果外部负载均衡器工作于传输层而不是工作于应用层的反向代理服务器，或者存在直接通过Ingress控制器接收客户端请求的需求，又期望它们能够提供HTTPS服务时，就应该配置TLS类型的Ingress资源。

将此类服务公开发布到互联网时，HTTPS服务用到的证书应由公信CA签署并颁发，用户遵循其相应流程准备好相关的数字证书即可。如果出于测试或内部使用之目的，那么也可以选择自制私有证书。openssl工具程序是用于生成自签证书的常用工具，这里使用它生成用于测试的私钥和自签证书：

```
~]$ openssl genrsa -out tls.key 2048
~]$ openssl req -new -x509 -key tls.key -out tls.crt \
    -subj /C=CN/ST=Beijing/L=Beijing/O=DevOps/CN=tomcat.ilinux.io -days 3650
```



注意 TLS Secret中包含的证书必须以tls.crt作为其键名，私钥文件必须以tls.key为键名，因此上面生成的私钥文件和证书文件名将直接保存为键名形式，以便于后面创建Secret对象时直接作为键名引用。

在Ingress控制器上配置HTTPS主机时，不能直接使用私钥和证书文件，而是要使用Secret资源对象来传递相关的数据。所以，接下来要根据私钥和证书生成用于配置TLS Ingress的Secret资源，在创建Ingress规则时由其将用到的Secret资源中的信息注入Ingress控制器的Pod对象中，用于为配置的HTTPS虚拟主机提供相应的私钥和证书。下面的命令会创建一个TLS类型名为tomcat-ingress-secret的Secret资源：

```
~]$ kubectl create secret tls tomcat-ingress-secret --cert=tls.crt --key=tls.
key -n testing
```

可使用下面的命令确认Secrets资源tomcat-ingress-secret创建成功完成:

```
~]$ kubectl get secrets tomcat-ingress-secret -n testing
NAME                                TYPE                      DATA  AGE
tomcat-ingress-secret              kubernetes.io/tls        2      20s
```

而后去定义创建TLS类型Ingress资源的配置清单。下面的配置清单通过spec.rules定义了一组转发规则，并通过spec.tls将此主机定义为了HTTPS类型的虚拟主机，用到的私钥和证书信息则来自于Secrets资源tomcat-ingress-secret:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: tomcat-ingress-tls
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - tomcat.ilinux.io
    secretName: tomcat-ingress-secret
  rules:
  - host: tomcat.ilinux.io
    http:
      paths:
      - path: /
        backend:
          serviceName: tomcat-svc
          servicePort: 80
```

运行资源创建命令完成Service资源的创建:

```
~]$ kubectl apply -f tomcat-ingress-tls.yaml
```

而后通过详细信息确认其创建成功完成，且已经正确关联到相应的tomcat-svc资源:

```
~]$ kubectl describe ingress tomcat-ingress-tls -n testing
Name:                tomcat-ingress-tls
Namespace:           testing
Address:
Default backend:     default-http-backend:80 (<none>)
```

```

TLS:
  tomcat-ingress-secret terminates tomcat.ilinux.io
Rules:
  Host                Path  Backends
  ----
  tomcat.ilinux.io    /    tomcat-svc:80 (<none>)
Annotations:
Events:
  Type    Reason    Age    From                                Message
  ----    -
  Normal  CREATE    31s    nginx-ingress-controller          Ingress testing/tomcat-ingress-
  tls

```

接下来即可通过Ingress控制器的前端Service资源的NodePort来访问此服务，在6.6.4节中，此Service资源以节点端口30443映射控制器的443端口。因此，这里使用Ingress中定义的主机名tomcat.ilinux.io:30443即可访问tomcat应用，其访问到的页面效果类似于图6-16中的内容。另外，也可以使用curl进行访问测试，只要对应的主机能够正确解析tomcat.ilinux.io主机名即可，例如，下面的测试命令及其输出表明，TLS类型的Ingress已然配置成功：

```

~]# curl -k -v https://tomcat.ilinux.io:30443/
* About to connect() to tomcat.ilinux.io port 30443 (#0)
*   Trying 172.16.0.66...
.....
* Server certificate:
*   subject: CN=tomcat.ilinux.io,O=DevOps,L=Beijing,ST=Beijing,C=CN
*   start date: Aug 23 07:05:31 2018 GMT
*   expire date: Aug 20 07:05:31 2028 GMT
*   common name: tomcat.ilinux.io
*   issuer: CN=tomcat.ilinux.io,O=DevOps,L=Beijing,ST=Beijing,C=CN
> GET / HTTP/1.1
.....
< HTTP/1.1 200 OK
< Server: nginx/1.13.9
< Date: Thu, 23 Aug 2018 07:37:40 GMT
.....

```

到此为止，实践配置目标已经全部达成。需要再次提醒的是，在实际使用中，在集群之外应该存在一个用于调度用户请求至各节点上Ingress控制器相关的NodePort的负载均衡器。如果不具有LBaaS的使用条件，用户也可以基于Nginx、Haproxy、LVS等手动构建，并通过Keepalived等解决方案实现其服务的高可用配置。

6.8 本章小结

本章重点讲解了Kubernetes的Service资源及其发布方式，具体如下。

- Service资源通过标签选择器为一组Pod资源创建一个统一的访问入口，其可将客户端请求代理调度至后端的Pod资源。

- Service资源是四层调度机制，默认调度算法为随机调度。

- Service的实现模式有三种：userspace、iptables和ipvs。

- Service共用四种类型：ClusterIP、NodePort、LoadBalancer和ExternalName，它们用于发布服务。

- Headless service是一种特殊的Service资源，可用于Pod发现。

- Ingress资源是发布Service资源的另一种方式，它需要结合Ingress控制器才能正常工作。

- Ingress Controller的实现方式除了Nginx之外，还有Envoy、HAProxy、Traefik等。

第7章 存储卷与数据持久化

应用程序在处理请求时，可根据其对当前请求的处理是否受影响于此前的请求，将应用划分为有状态应用和无状态应用两种。微服务体系中，各种应用均被拆分成了众多微服务或更小的应用模块，因此往往会存在为数不少的有状态应用，当然，也会存在数量可观的无状态应用。而对于有状态应用来说，数据持久化几乎是必然之需。

Kubernetes提供的存储卷（Volume）属于Pod资源级别，共享于Pod内的所有容器，可用于在容器的文件系统之外存储应用程序的相关数据，甚至还可独立于Pod的生命周期之外实现数据持久化。本章主要介绍Kubernetes系统之上的主流存储卷类型及其应用。

7.1 存储卷概述

Pod本身具有生命周期，故其内部运行的容器及其相关数据自身均无法持久存在。**Docker**支持配置容器使用存储卷将数据持久存储于容器自身文件系统之外的存储空间中，它们可以是节点文件系统或网络文件系统之上的存储空间。相应地，**Kubernetes**也支持类似的存储卷功能，不过，其存储卷是与**Pod**资源绑定而非容器。简单来说，存储卷是定义在**Pod**资源之上、可被其内部的所有容器挂载的共享目录，它关联至某外部的存储设备之上的存储空间，从而独立于容器自身的文件系统，而数据是否具有持久能力则取决于存储卷自身是否支持持久机制。**Pod**、容器与存储卷的关系如图7-1所示。

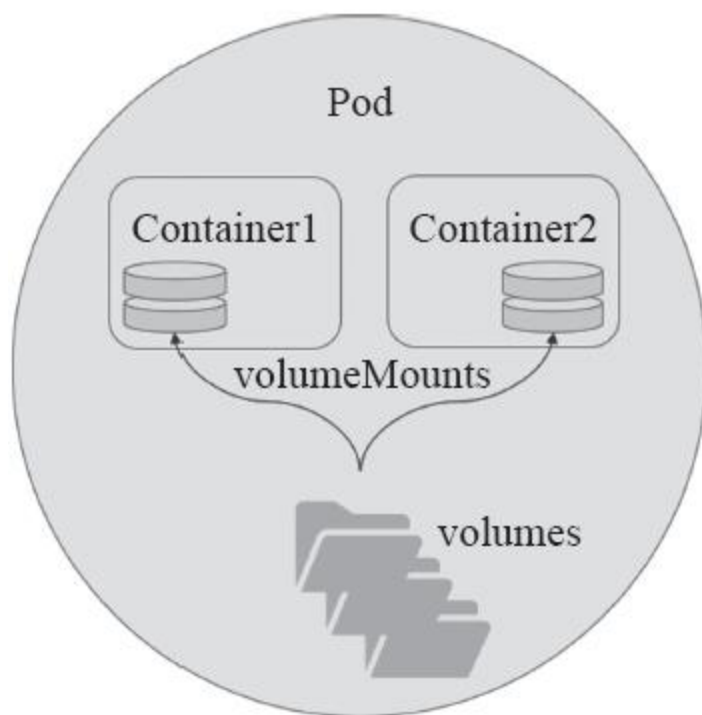


图7-1 Pod、容器与存储卷

7.1.1 Kubernetes支持的存储卷类型

Kubernetes支持非常丰富的存储卷类型，包括本地存储（节点）和网络存储系统中的诸多存储机制，甚至还支持Secret和ConfigMap这样的特殊存储资源。对于Pod来说，卷类型主要是为关联相关的存储系统时提供相关的配置参数，例如，关关节点本地的存储目录与关联GlusterFS存储系统所需要的配置参数差异巨大，因此指定存储卷类型时也就限定了其关联到的后端存储设备。目前，Kubernetes支持的存储卷包含以下这些类型。

- emptyDir

- hostPath

- nfs

- fc

- iscsi

- flocker

- Glusterfs

- rbd

- cephfs

- cinder

- awsElasticBlockStore

- gcePersistentDisk

- azureDisk

- azureFile

- gitRepo
- downwardAPI
- ConfigMap
- secret
- projected
- name
- persistentVolumeClaim
- downwardAPI
- vsphereVolume
- quobyte
- portworxVolume
- photonPersistentDisk
- scaleIO
- flexVolume
- storageOS
- local

上述类型中，emptyDir与hostPath属于节点级别的卷类型，emptyDir的生命周期与Pod资源相同，而使用了hostPath卷的Pod一旦被重新调度至其他节点，那么它将无法再使用此前的数据。因此，这两种类型都不具有持久性。要想使用持久类型的存储卷，就得使用网络存储系统，如NFS、Ceph、GlusterFS等，或者云端存储，如gcePersistentDisk、awsElasticBlockStore等。

然而，网络存储系统通常都不太容易使用，有的甚至很复杂，以至于对大多数用户来说它是一个难以逾越的障碍。Kubernetes为此专门设计了一种集群级别的资源PersistentVolume（简称PV），它借由管理员配置存储系统，而后由用户通过“persistentVolumeClaim”（简称PVC）存储卷直接申请使用的机制大大简化了终端存储用户的配置过程，有效降低了使用难度。

再者，Secret和ConfigMap算得上是两种特殊的卷类型。

1) Secret用于向Pod传递敏感信息，如密码、私钥、证书文件等，这些信息如果直接定义在镜像中很容易导致泄露，有了Secret资源，用户可以将这些信息存储于集群中而后由Pod进行挂载，从而实现将敏感数据与系统解耦。

2) ConfigMap资源则用于向Pod注入非敏感数据，使用时，用户将数据直接存储于ConfigMap对象中，而后直接在Pod中使用ConfigMap卷引用它即可，它可以帮助实现容器配置文件集中化定义和管理。

另外，Kubernetes自1.9版本起对存储的支持做了进一步的增强，引入了容器存储接口（Container Storage Interface, CSI）的一套alpha实现版本，其能够将插件的安装流程简化至与创建Pod相当，并允许第三方存储供应商在无须修改Kubernetes代码库的前提下提供自己的解决方案。因此，Kubernetes系统支持的卷存储机制必将进一步增强。

7.1.2 存储卷的使用方式

在Pod中定义使用存储卷的配置由两部分组成：一是通过`.spec.volumes`字段定义在Pod之上的存储卷列表，其支持使用多种不同类型的存储卷且配置参数差别很大；另一个是通过`.spec.containers.volumeMounts`字段在容器上定义的存储卷挂载列表，它只能挂载当前Pod资源中定义的具体存储卷，当然，也可以不挂载任何存储卷。如图7-1所示。

在Pod级别定义存储卷时，`.spec.volumes`字段的值是对象列表格式，每个对象为一个存储卷的定义，它由存储卷名称（`.spec.volumes.name<String>`）或存储卷对象（`.spec.volumes.VOL_TYPE<Object>`）组成，其中VOL_TYPE是使用的存储卷类型名称，它的内嵌字段随类型的不同而不同。下面的资源清单片段定义了由两个存储卷组成的卷列表，一个是`emptyDir`类型，一个是`gitRepo`类型：

```
spec:
  ...
  volumes:
  - name: logdata
    emptyDir: {}
  - name: example
    gitRepo:
      repository: https://github.com/iKubernetes/k8s_book.git
      revision: master
      directory: .
```

定义好的存储卷可由当前Pod资源内的各容器进行挂载。事实上，也只有多个容器挂载同一个存储卷时，“共享”才有了具体的意义。当Pod中只有一个容器时，使用存储卷的目的通常在于数据持久化。`.spec.containers.volumeMounts`字段的值也是对象列表格式，由一到多个存储卷挂载定义组成。无论何种类型的存储卷，它们的挂载格式基本上都是相同的，下面的代码段是在容器中定义挂载卷时的通用语法形式：

```
spec:
  ...
  containers:
```

```
- name: <String>
...
volumeMounts:
- name <string> -required-
  mountPath <string> -required-
  readOnly <boolean>
  subPath <string>
  mountPropagation <string>
```

其中各字段的意义及使用要求具体如下。

·**name<string>**: 指定要挂载的存储的名称，必选字段。

·**mountPath<string>**: 挂载点路径，容器文件系统上的路径，必选字段。

·**readOnly<boolean>**: 是否挂载为只读卷。

·**subPath<string>**: 挂载存储卷时使用的子路径，即在**mountPath**指定的路径下使用一个子路径作为其挂载点。

下面是一个挂载示例，容器**myapp**将**logdata**存储卷挂载于**/var/log/myapp**，将**example**挂载到**/webdata/example**目录：

```
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v7
    volumeMounts:
    - name: logdata
      mountPath: /var/log/myapp/
    - name: example
      mountPath: /webdata/example/
```

存储卷的定义基本相似，因此本章后面的篇幅将重点放在介绍主流存储卷类型的配置及使用方式，而且出于保持示例简洁及节约篇幅的目的，本章的示例主要以自主式**Pod**资源为主。

7.2 临时存储卷

Kubernetes支持存储卷类型中，`emptyDir`存储卷的生命周期与其所属的Pod对象相同，它无法脱离Pod对象的生命周期提供数据存储功能，因此`emptyDir`通常仅用于数据缓存或临时存储。不过，基于`emptyDir`构建的`gitRepo`存储卷可以在Pod对象的生命周期起始时从相应的Git仓库中复制相应的数据文件到底层的`emptyDir`中，从而使得它具有了一定意义上的持久性。

7.2.1 emptyDir存储卷

emptyDir存储卷是Pod对象生命周期中的一个临时目录，类似于Docker上的“docker挂载卷”，在Pod对象启动时即被创建，而在Pod对象被移除时会被一并删除。不具有持久能力的emptyDir存储卷只能用于某些特殊场景中，例如，同一Pod内的多个容器间文件的共享，或者作为容器数据的临时存储目录用于数据缓存系统等。

emptyDir存储卷则定义于.spec.volumes.emptyDir嵌套字段中，可用字段主要包含两个，具体如下。

- medium: 此目录所在的存储介质的类型，可取值为“default”或“Memory”，默认为default，表示使用节点的默认存储介质；“Memory”表示使用基于RAM的临时文件系统tmpfs，空间受限于内存，但性能非常好，通常用于为容器中的应用提供缓存空间。

- sizeLimit: 当前存储卷的空间限额，默认值为nil，表示不限制；不过，在medium字段值为“Memory”时建议务必定义此限额。

下面是一个使用了emptyDir存储卷的简单示例，它保存于vol-emptydir.yaml配置文件中：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-emptydir-pod
spec:
  volumes:
    - name: html
      emptyDir: {}
  containers:
    - name: nginx
      image: nginx:1.12-alpine
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
    - name: pagegen
      image: alpine
      volumeMounts:
        - name: html
          mountPath: /html
      command: ["/bin/sh", "-c"]
      args:
        - while true; do
```

```
    echo $(hostname) $(date) >> /html/index.html;
    sleep 10;
done
```

上面的示例中定义的存储卷名称为html，挂载于容器nginx的/usr/share/nginx/html目录，以及容器pagegen的/html目录。容器pagegen每隔10秒向存储卷上的index.html文件中追加一行信息，而容器nginx中的nginx进程则以其为站点主页，如图7-2所示。

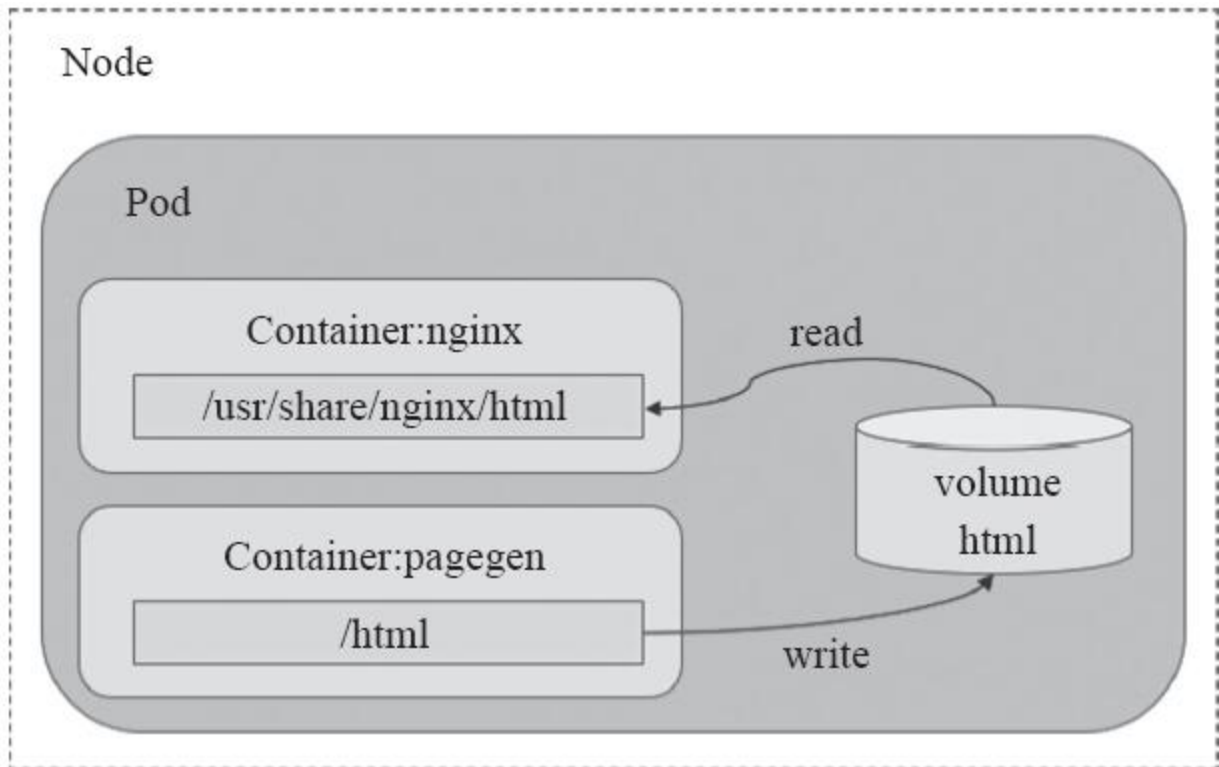


图7-2 存储卷使用示意图

Pod资源的详细信息中会显示存储卷的相关状态，包括其是否创建成功（在Events字段中输出）、相关的类型及参数（在Volumes字段中输出）以及容器中的挂载状态等信息（在Containers字段中输出）。如下面的命令结果所示：

```
~]$ kubectl describe pods vol-emptydir-pod
Name:          vol-emptydir-pod
.....
Containers:
  nginx:
.....
Mounts:
```

```
    /usr/share/nginx/html from html (rw)
pagegen:
.....
    Mounts:
        /html from html (rw)
.....
Volumes:
    html:
        Type:     EmptyDir (a temporary directory that shares a pod's lifetime)
        Medium:
.....
Events:
  Type      Reason            Age   From          Message
  ----      -
Normal     Scheduled         10s   default-scheduler Successfully assigned vol-emptydir
to node03.ilinux.io
Normal     SuccessfulMountVolume 10s   kubelet, node03.ilinux.io MountVolume.
Setup succeeded for volume "html"
```

而后，可以为其创建**Service**资源并进行访问测试，或者在集群中直接对**Pod**的地址发起访问请求，以测试两个容器通过**emptyDir**卷共享数据的结果状态：

```
~]$ curl 10.1.3.106
vol-emptydir Wed Mar 7 01:10:56 UTC 2018
```

作为边车（**sidecar**）的容器**paggen**，其每隔10秒生成一行信息追加到存储卷上的**index.html**文件中，因此，通过主容器**nginx**的应用访问到的内容也会处于不停的变动中。另外，**emptyDir**存储卷也可以基于**RAM**创建**tmpfs**文件系统的存储卷，常用于为容器的应用提供高性能缓存，下面是一个配置示例：

```
volumes:
- name: cache
  emptyDir:
    medium: Memory
```

emptyDir卷简单易用，但仅能用于临时存储。另外还存在一些类型的存储卷建构在**emptyDir**之上，并额外提供了它所没有的功能，例如，将于7.2.2节介绍的**gitRepo**存储卷。

7.2.2 gitRepo存储卷

gitRepo存储卷可以看作是emptyDir存储卷的一种实际应用，使用该存储卷的Pod资源可以通过挂载目录访问指定的代码仓库中的数据。使用gitRepo存储卷的Pod资源在创建时，会首先创建一个空目录（emptyDir）并克隆（clone）一份指定的Git仓库中的数据至该目录，而后再创建容器并挂载该存储卷。

定义gitRepo类型的存储卷时，其可嵌套使用的字段具体包含如下三个。

·repository<string>：Git仓库的URL，必选字段。

·directory<string>：目标目录名称，名称中不能包含“.”字符；“.”表示将仓库中的数据直接复制到卷目录中，否则，即为复制到卷目录中以用户指定的字符串为名称的子目录中。

·revision<string>：特定revision的提交哈希码。



注意 使用gitRepo存储卷的Pod资源运行的工作节点上必须安装有Git程序，否则克隆仓库的操作将无从完成。另外，自Kubernetes 1.12起，gitRepo存储卷已经被废弃。

下面示例（vol-gitrepo.yaml配置文件）中的Pod资源在创建时，首先会创建一个空目录，将指定的Git仓库https://github.com/iKubernetes/k8s_book.git中的数据复制一份直接保存于此目录中，而后将此目录创建为存储卷html，最后由容器Nginx将此存储卷挂载于/usr/share/nginx/html目录上：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-gitrepo-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.12-alpine
    volumeMounts:
```

```
- name: html
  mountPath: /usr/share/nginx/html
volumes:
- name: html
  gitRepo:
    repository: https://github.com/iKubernetes/k8s_book.git
    directory: .
    revision: "master"
```

访问此Pod资源中的Nginx服务即可看到其来自于Git仓库中的页面资源。不过，gitRepo存储卷在其创建完成后不会再与指定的仓库执行同步操作，这就意味着在Pod资源运行期间，如果仓库中的数据发生了变化，那么gitRepo存储卷不会同步到这些内容。当然，此时可以为Pod资源创建一个专用的边车容器用于执行此类的同步操作，尤其是数据来源于私有仓库时，通过边车容器完成其复制就更为必要。

gitRepo存储卷建构于emptyDir存储卷之上，它的生命周期与隶属的Pod对象相同，因此使用时不建议在此类存储卷中保存由容器生成的重要数据。另外，自Kubernetes 1.12版起，gitRepo存储卷已经被废弃，所以在之后的版本中若要使用它配置Pod对象，建议读者借助初始化容器（InitContainer）将仓库中的数据复制到emptyDir存储卷上，并在主容器中使用此存储卷。

7.3 节点存储卷hostPath

hostPath类型的存储卷是指将工作节点上某文件系统的目录或文件挂载于**Pod**中的一种存储卷，它可独立于**Pod**资源的生命周期，因而具有持久性。但它是工作节点本地的存储空间，仅适用于特定情况下的存储卷使用需求，例如，将工作节点上的文件系统关联为**Pod**的存储卷，从而使得容器访问节点文件系统上的数据。这一点在运行有管理任务的系统级**Pod**资源需要访问节点上的文件时尤为有用。

配置**hostPath**存储卷的嵌套字段共有两个：一个是用于指定工作节点上的目录路径的必选字段**path**；另一个是指定存储卷类型的**type**，它支持使用的卷类型包含如下几种。

- DirectoryOrCreate**：指定的路径不存时自动将其创建为权限是0755的空目录，属主属组均为**kubelet**。

- Directory**：必须存在的目录路径。

- FileOrCreate**：指定的路径不存时自动将其创建为权限是0644的空文件，属主和属组同是**kubelet**。

- File**：必须存在的文件路径。

- Socket**：必须存在的Socket文件路径。

- CharDevice**：必须存在的字符设备文件路径。

- BlockDevice**：必须存在的块设备文件路径。

下面是定义在**vol-hostpath.yaml**配置文件中的**Pod**资源，它运行着日志收集代理应用**filebeat**，负责收集工作节点及容器相关的日志信息发往**Redis**服务器，它使用了三个**hostPath**类型的存储卷：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-hostpath-pod
spec:
  containers:
```

```
- name: filebeat
  image: ikubernetes/filebeat:5.6.7-alpine
  env:
    - name: REDIS_HOST
      value: redis.ilinux.io:6379
    - name: LOG_LEVEL
      value: info
  volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: socket
      mountPath: /var/run/docker.sock
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
    - name: socket
      hostPath:
        path: /var/run/docker.sock
```

这类Pod资源通常受控于daemonset类型的Pod控制器，它运行于集群中的每个工作节点之上，负责收集工作节点上系统级的相关数据，因此使用hostPath存储卷也是理所应当的。读者在创建上述Pod资源时，如果有可用的Redis服务器，则可通过环境变量REDIS_HOST传递给Pod资源，待其Ready之后即可通过Redis服务器查看到由其发送的日志信息。在filebeat的应用架构中，这些日志信息会发往Elasticsearch，并通过Kibana进行展示。

另外，使用hostPath存储卷时需要注意到，不同节点上的文件或许并不完全相同，于是，那些要求事先必须存在的文件或目录的满足状态也可能会有所不同；另外，基于资源可用状态的调度器调度Pod时，hostPath资源的可用性状态不会被考虑在内；再者，在节点中创建的文件或目录默认仅有root可写，若期望容器内的进程拥有写权限，则要么将它运行为特权容器，要么修改节点上目录路径的权限。

那些并非执行系统级管理任务的且不受控于Daemonset控制器的无状态应用在Pod资源被重新调度至其他节点运行时，此前创建的文件或目录大多都不会存在。因此，hostPath存储卷虽然能持久保存数据，但对于被调度器按需调度的应用来说并不适用，这时需要用到的是独立于集群节点的持久性存储卷，即网络存储卷。

7.4 网络存储卷

如前所述，Kubernetes拥有众多类型的用于适配专用存储系统的网络存储卷。这类存储卷包括传统的NAS或SAN设备（如NFS、iSCSI、fc）、分布式存储（如GlusterFS、RBD）、云端存储（如gcePersistentDisk、azureDisk、cinder和awsElasticBlockStore）以及建构在各类存储系统之上的抽象管理层（如flocker、portworxVolume和vsphereVolume）等。

7.4.1 NFS存储卷

NFS即网络文件系统（Network File System），它是一种分布式文件系统协议，最初是由Sun Microsystems公司开发的类Unix操作系统之上的一款经典的网络存储方案，其功能旨在允许客户端主机可以像访问本地存储一样通过网络访问服务器端文件。作为一种由内核原生支持的网络文件系统，具有Linux系统使用经验的读者大多数都应该对NFS有一定的使用经验。

Kubernetes的NFS存储卷用于将某事先存在的NFS服务器上导出（export）的存储空间挂载到Pod中以供容器使用。与emptyDir不同的是，NFS存储卷在Pod对象终止后仅是被卸载而非删除。另外，NFS是文件系统级共享服务，它支持同时存在的多路挂载请求。定义NFS存储卷时，常用到以下字段。

- server<string>：NFS服务器的IP地址或主机名，必选字段。
- path<string>：NFS服务器导出（共享）的文件系统路径，必选字段。
- readOnly<boolean>：是否以只读方式挂载，默认为false。

Redis（REmote DIctionary Server）是一个著名的高性能key-value存储系统，应用非常广泛，将其部署运行于Kubernetes系统之上时，需要持久化存储卷的支持。下面是简单使用Redis的一个示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-nfs-pod
  labels:
    app: redis
spec:
  containers:
  - name: redis
    image: redis:4-alpine
    ports:
    - containerPort: 6379
      name: redisport
    volumeMounts:
    - mountPath: /data
      name: redisdata
```

```
volumes:
- name: redisdata
  nfs:
    server: nfs.ilinux.io
    path: /data/redis
    readOnly: false
```

上面的示例定义在资源配置文件vol-nfs.yaml中，其中的Pod资源拥有一个关联至NFS服务器nfs.ilinux.io的存储卷，Redis容器将其挂载于/data目录上，它是运行于容器中的redis-server数据的持久保存位置。



提示 这里应确保事先要存在一个名为nfs.ilinux.io的NFS服务器，其输出了/data/redis目录，并授权给了Kubernetes集群中的节点访问。主机和目录都可以按需要进行调整。

资源创建完成后，可通过其命令客户端redis-cli创建测试数据，并手动触发其同步于存储系统中，下面加粗部分的字体为要执行的Redis命令：

```
~]$ kubectl exec -it vol-nfs-pod redis-cli
127.0.0.1:6379> set mykey "hello ilinux.io"
OK
127.0.0.1:6379> get mykey
"hello ilinux.io"
127.0.0.1:6379> BGSAVE
Background saving started
127.0.0.1:6379> exit
```

为了测试其数据持久化效果，下面删除Pod资源vol-nfs-pod，并于再次重建后检测数据是否依然能够访问：

```
~]$ kubectl delete pods vol-nfs-pod
pod "vol-nfs-pod" deleted
~]$ kubectl apply -f vol-nfs.yaml
pod "vol-nfs-pod" created
```

待其重建完成后，通过再一次创建的Pod资源的详细信息，我们可以观察到它挂载使用NFS存储卷的相关信息。接下来再次检查redis-server中是否还保存有此前存储的数据：

```
~]$ kubectl exec -it vol-nfs-pod redis-cli
127.0.0.1:6379> get mykey
"hello ilinux.io"
127.0.0.1:6379>
```

从上面的命令结果中可以看出，此前创建的键`mykey`及其数据在Pod资源重建后依然存在，这表明在删除Pod资源时，其关联的外部存储卷并不会被一同删除。如果需要清除此类的数据，需要用户通过存储系统的管理接口手动进行。

7.4.2 RBD存储卷

Ceph是一个专注于分布式的、弹性可扩展的、高可靠的、性能优异的存储系统平台，同时支持提供块设备、文件系统和REST三种存储接口。它是一个高度可配置的系统，并提供了一个命令行界面用于监视和控制其存储集群。Ceph还包含鉴证和授权功能，可兼容多种存储网关接口，如OpenStack Swift和Amazon S3。Kubernetes也支持通过RBD卷类型使用Ceph存储系统为Pod提供存储卷。要配置Pod资源使用RBD存储卷，需要事先满足如下几个前提条件。

- 存在某可用的Ceph RBD存储集群，否则就需要创建一个。
- 在Ceph RBD集群中创建一个能满足Pod资源数据存储需要的存储映像（image）。
- 在Kubernetes集群内的各节点上安装Ceph客户端程序包（ceph-common）。

在配置RBD类型的存储卷时，需要指定要连接的目标服务器和认证信息等，这一点通常使用以下嵌套字段进行定义。

- `monitors<[]string>`: Ceph存储监视器，逗号分隔的字符串列表；必选字段。
- `image<string>`: rados image的名称，必选字段。
- `pool<string>`: rados存储池名称，默认为RBD。
- `user<string>`: rados用户名，默认为admin。
- `keyring<string>`: RBD用户认证时使用的keyring文件路径，默认为/etc/ceph/keyring。
- `secretRef<Object>`: RBD用户认证时使用的保存有相应认证信息的Secret对象，会覆盖由keyring字段提供的密钥信息。
- `readOnly<string>`: 是否以只读的方式进行访问。

·fsType: 要挂载的存储卷的文件系统类型，至少应该是节点操作系统支持的文件系统，如ext4、xfs、ntfs等，默认为ext4。

下面是一个定义在vol-rbd.yaml配置文件中使用的RBD存储卷的Pod资源示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-rbd-pod
spec:
  containers:
    - name: redis
      image: redis:4-alpine
      ports:
        - containerPort: 6379
          name: redisport
      volumeMounts:
        - mountPath: /data
          name: redis-rbd-vol
  volumes:
    - name: redis-rbd-vol
      rbd:
        monitors:
          - '172.16.0.56:6789'
          - '172.16.0.57:6789'
          - '172.16.0.58:6789'
        pool: kube
        image: redis
        fsType: ext4
        readOnly: false
        user: admin
        secretRef:
          name: ceph-secret
```

此示例依赖于事先存在的一个Ceph存储集群，这里假设其监视器的地址为172.16.0.56、172.16.0.57和172.16.0.58三个主机IP，并且集群上的存储池kube中存在创建好的映像Redis，此映像拥有ext4文件系统。Ceph客户端访问集群时需要事先完成认证之后才能进行后续的访问操作，此示例上，其认证信息保存于名为ceph-secret的Secret资源对象中。示例所实现的逻辑架构如图7-3所示。

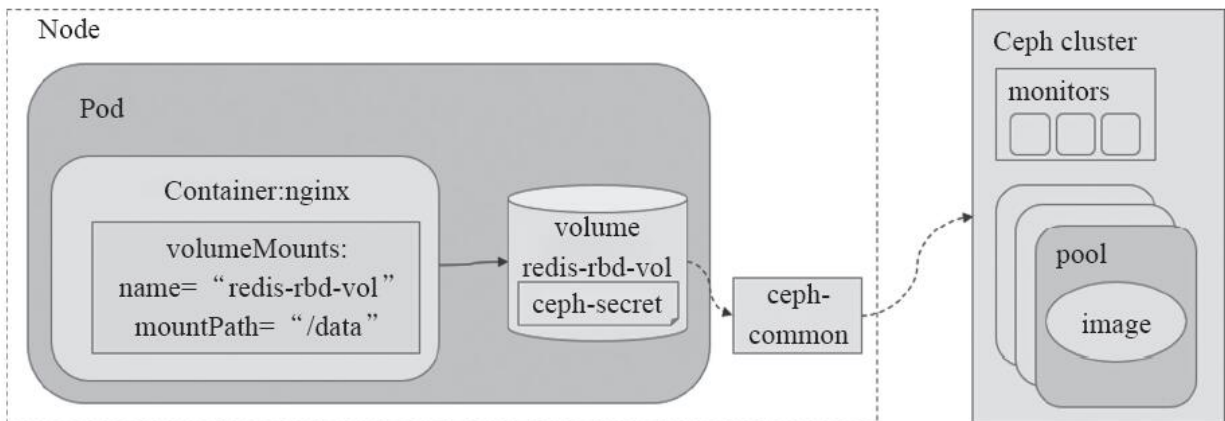



图7-3 RBD存储卷

 **提示** 此配置示例依赖于一个可用的Ceph集群，且上述配置示例中的部分参数需要参照实际环境进行修改。

7.4.3 GlusterFS存储卷

GlusterFS (Gluster File System) 是一个开源的分布式文件系统，是水平扩展存储解决方案Gluster的核心，具有强大的横向扩展能力，GlusterFS通过扩展能够支持数PB存储容量和处理数千客户端。GlusterFS借助TCP/IP或InfiniBand RDMA网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。另外，GlusterFS基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能，是另一种流行的分布式存储解决方案。要配置Pod资源使用GlusterFS存储卷，需要事先满足以下前提条件。

- 1) 存在某可用的GlusterFS存储集群，否则就要创建一个。
- 2) 在GlusterFS集群中创建一个能满足Pod资源数据存储需要的卷。
- 3) 在Kubernetes集群内的各节点上安装GlusterFS客户端程序包 (glusterfs和glusterfs-fuse)。

另外，若要基于GlusterFS使用存储卷的动态供给机制（请参考7.5.6节），还需要事先部署heketi，它用于为GlusterFS集群提供RESTful风格的管理接口。Gluster存储集群及heketi的配置示例请参考附录B。

定义Pod资源使用GlusterFS类型的存储卷时，常用的配置字段包含如下几个。

·`endpoints<string>`: Endpoints资源的名称，此资源需要事先存在，用于提供Gluster集群的部分节点信息作为其访问入口；必选字段。

·`path<string>`: 用到的GlusterFS集群的卷路径，如kube-redis；必选字段。

·`readOnly<boolean>`: 是否为只读卷。

下面是一个定义在vol-glusterfs.yaml配置文件中的Pod资源示例，它使用了GlusterFS存储卷持久保存应用数据。它通过glusterfs-endpoints资源中定义的GlusterFS集群节点信息接入集群，并以kube-redis卷作为Pod资源的存储卷。glusterfs-endpoints资源需要在Kubernetes集群中事先创建，而kube-redis则需要事先创建于Gluster集群：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-glusterfs-pod
  labels:
    app: redis
spec:
  containers:
  - name: redis
    image: redis:alpine
    ports:
    - containerPort: 6379
      name: redisport
    volumeMounts:
    - mountPath: /data
      name: redisdata
  volumes:
  - name: redisdata
    glusterfs:
      endpoints: glusterfs-endpoints
      path: kube-redis
      readOnly: false
```

用于访问Gluster集群的相关节点信息要事先保存于某特定的Endpoints资源中，例如上面示例中调用的glusterfs-endpoints。此类的Endpoints资源可由用户根据实际需求手动创建，例如，下面的保存于glusterfs-endpoints.yaml文件中的资源示例中定义了三个接入相关的Gluster存储集群的节点gfs01.ilinux.io、gfs02.ilinux.io和gfs03.ilinux.io，其中的端口信息仅为满足Endpoints资源的必选字段要求，因此其值可以随意填写：

```
apiVersion: v1
kind: Endpoints
metadata:
  name: glusterfs-endpoints
subsets:
  - addresses:
    - ip: gfs01.ilinux.io
    ports:
    - port: 24007
      name: glusterd
```

```
- addresses:
  - ip: gfs02.ilinux.io
  ports:
    - port: 24007
      name: glusterd
- addresses:
  - ip: gfs03.ilinux.io
  ports:
    - port: 24007
      name: glusterd
```

首先创建Endpoints资源glusterfs-endpoints，而后再创建Pod资源vol-glusterfs-pod，即可测试其数据持久存储的效果。

7.4.4 Cinder存储卷

Cinder是OpenStack Block Storage的项目名称，用来为虚拟机（VM）实例提供持久块存储。Cinder通过驱动架构支持多种后端（back-end）存储方式，包括LVM、NFS、Ceph和其他诸如EMC、IBM等商业存储产品和方案，其提供了调度度来调度卷创建的请求，能合理优化存储资源的分配，而且还拥有REST API。将Kubernetes集群部署于OpenStack构建的IaaS环境中时，Cinder的块存储功能可为Pod资源提供外部持久存储的有效方式。

在Pod资源上定义使用Cinder存储卷时，其可用的嵌套字段包含如下几个。

- volumeID<string>：用于标识Cinder中的存储卷的卷标识符，必选字段。

- readOnly<boolean>：是否以只读方式访问。

fsType：要挂载的存储卷的文件系统类型，至少应该是节点操作系统支持的文件系统，如ext4、xfs、ntfs等，默认为“ext4”。

下面的资源清单是定义在vol-cinder.yaml文件中的使用示例，假设在OpenStack环境中已创建好的Cinder卷“e2b8d2f7-wece-90d1-a505-4acf607a90bc”可用：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-cinder-pod
spec:
  containers:
  - image: mysql
    name: mysql
    args:
      - "--ignore-db-dir"
      - "lost+found"
    env:
      - name: MYSQL_ROOT_PASSWORD
        value: YOUR_PASS
    ports:
      - containerPort: 3306
        name: mysqlport
```

```
    volumeMounts:
      - name: mysqldata
        mountPath: /var/lib/mysql
  volumes:
    - name: mysqldata
      cinder:
        volumeID: e2b8d2f7-wece-90d1-a505-4acf607a90bc
        fsType: ext4
```

配置可用的系统环境和存储资源时，将其匹配于资源清单文件中即可完成Pod资源创建。另外，Kubernetes所支持各类持久存储卷其配置使用方式各有不同，鉴于篇幅有限，这里不再一一列举其使用方式。

7.5 持久存储卷

通过前面使用持久存储卷（**Persistent Volume**）的示例可知，**Kubernetes**用户必须要清晰了解所用到的网络存储系统的访问细节才能完成存储卷相关的配置任务，例如，**NFS**存储卷的**server**和**path**字段的配置就依赖于服务器地址和共享目录路径。这与**Kubernetes**的向用户和开发隐藏底层架构的目标有所背离，对存储资源的使用最好也能像使用计算资源一样，用户和开发人员无须了解**Pod**资源究竟运行于哪个节点，也无须了解存储系统是什么设备以及位于何处。为此，**Kubernetes**的**PersistentVolume**子系统在用户与管理员之间添加了一个抽象层，从而使得存储系统的使用和管理职能互相解耦，如图7-4所示。

PersistentVolume（**PV**）是指由集群管理员配置提供的某存储系统上的一段存储空间，它是对底层共享存储的抽象，将共享存储作为一种可由用户申请使用的资源，实现了“存储消费”机制。通过存储插件机制，**PV**支持使用多种网络存储系统或云端存储等多种后端存储系统，例如，前面使用的**NFS**、**RBD**和**Cinder**等。**PV**是集群级别的资源，不属于任何名称空间，用户对**PV**资源的使用需要通过**PersistentVolumeClaim**（**PVC**）提出的使用申请（或称为声明）来完成绑定，是**PV**资源的消费者，它向**PV**申请特定大小的空间及访问模式（如**rw**或**ro**），从而创建出**PVC**存储卷，而后再由**Pod**资源通过**PersistentVolumeClaim**存储卷关联使用，如图7-4所示。

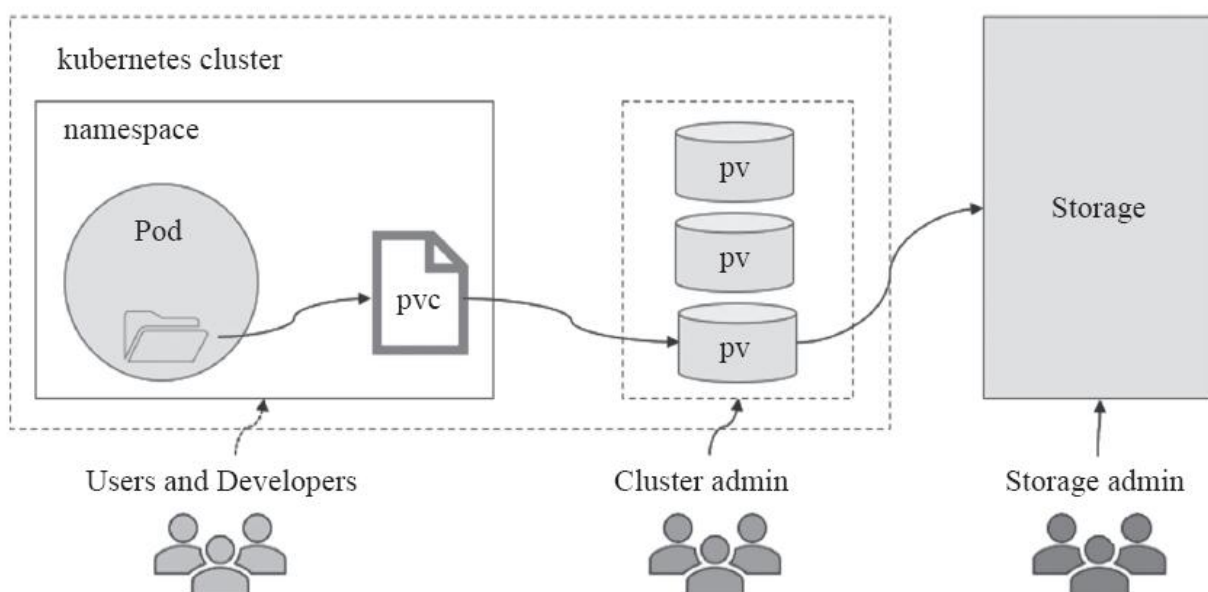


图7-4 Pod存储卷、PVC、PV及存储设备的调用关系

尽管PVC使得用户可以以抽象的方式访问存储资源，但很多时候还是会涉及PV的不少属性，例如，用于不同场景时设置的性能参数等。为此，集群管理员不得不通过多种方式提供多种不同的PV以满足用户不同的使用需求，两者衔接上的偏差必然会导致用户的需求无法全部及时有效地得到满足。Kubernetes自1.4版起引入了一个新的资源对象StorageClass，可用于将存储资源定义为具有显著特性的类别

（Class）而不是具体的PV，例如“fast”“slow”或“glod”“silver”“bronze”等。用户通过PVC直接向意向的类别发出申请，匹配由管理员事先创建的PV，或者由其按需为用户动态创建PV，这样做甚至免去了需要事先创建PV的过程。

PV对存储系统的支持可通过其插件来实现，目前，Kubernetes支持如下类型的插件。

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- AzureDisk
- FC（Fibre Channel）**
- FlexVolume
- Flocker
- NFS
- iSCSI
- RBD（Ceph Block Device）
- CephFS
- Cinder（OpenStack block storage）

- Glusterfs
- VsphereVolume
- Quobyte Volumes
- HostPath
- VMware Photon
- Portworx Volumes
- ScaleIO Volumes
- StorageOS

7.5.1 创建PV

PersistentVolume Spec主要支持以下几个通用字段，用于定义PV的容量、访问模式和回收策略。

1) Capacity: 当前PV的容量；目前，Capacity仅支持空间设定，将来应该还可以指定IOPS和throughput。

2) 访问模式：尽管在PV层看起来并无差别，但存储设备支持及启用的功能特性却可能不尽相同。例如NFS存储支持多客户端同时挂载及读写操作，但也可能是在共享时仅启用了只读操作，其他存储系统也存在类似的可配置特性。因此，PV底层的设备或许存在其特有的访问模式，用户使用时必须在其特性范围内设定其功能，具体如图7-5所示。

·ReadWriteOnce: 仅可被单个节点读写挂载；命令行中简写为RWO。

·ReadOnlyMany: 可被多个节点同时只读挂载；命令行中简写为ROX。

·ReadWriteMany: 可被多个节点同时读写挂载；命令行中简写为RWX。

3) persistentVolumeReclaimPolicy: PV空间被释放时的处理机制；可用类型仅为Retain（默认）、Recycle或删除Delete，具体说明如下。

·Retain: 保持不动，由管理员随后手动回收。

·Recycle: 空间回收，即删除存储卷目录下的所有文件（包括子目录和隐藏文件），目前仅NFS和hostPath支持此操作。

·Delete: 删除存储卷，仅部分云端存储系统支持，如AWS EBS、GCE PD、Azure Disk和Cinder。

4) volumeMode: 卷模型，用于指定此卷可被用作文件系统还是裸格式的块设备；默认为Filesystem。

- 5) storageClassName: 当前PV所属的StorageClass的名称；默认为空值，即不属于任何StorageClass。
- 6) mountOptions: 挂载选项组成的列表，如ro、soft和hard等。

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
FC	✓	✓	-
FlexVolume	✓	✓	-
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
PhotonPersistentDisk	✓	-	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	- (works when pods are collocated)
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

图7-5 各PV支持的访问模式

下面的资源清单配置示例中定义了一个使用NFS存储后端的PV，空间大小为10GB，支持多路的读写操作。待后端存储系统满足需求时，即可进行如下PV资源的创建：

```
kind: PersistentVolume
metadata:
  name: pv-nfs-0001
  labels:
    release: stable
spec:
```

```
capacity:
  storage: 5Gi
volumeMode: Filesystem
accessModes:
  - ReadWriteMany
persistentVolumeReclaimPolicy: Recycle
storageClassName: slow
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: "/webdata/htdocs"
  server: nfs.ilinux.io
```

创建完成后，可以看到其状态为“Available”，即“可用”状态，表示目前尚未被PVC资源所“绑定”：

```
~]$ kubectl get pv pv-nfs-0001 -o custom-columns=NAME:metadata.name,STATUS:status.
phase
NAME      STATUS
pv-nfs-0001  Available
```

下面是另一个PV资源的配置清单，它使用RBD存储后端，空间大小为2GB，仅支持单个客户端的读写访问。将RBD相关属性设定为匹配实际的环境需求，例如在Ceph集群中创建映像pv-rbd-0001，大小为2GB，并在映射后进行映像文件格式化，随后即可创建如下的PV资源：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-rbd-0001
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  rbd:
    monitors:
      - ceph-mon01.ilinux.io:6789
      - ceph-mon02.ilinux.io:6789
      - ceph-mon03.ilinux.io:6789
    pool: kube
    image: pv-rbd-0001
    user: admin
    secretRef:
      name: ceph-secret
    fsType: ext4
    readOnly: false
  persistentVolumeReclaimPolicy: Retain
```

使用资源的查看命令可列出**PV**资源的相关信息。创建完成的**PV**资源可能处于下列四种状态中的某一种，它们代表着**PV**资源生命周期中的各个阶段。

- Available**: 可用状态的自由资源，尚未被**PVC**绑定。
- Bound**: 已经绑定至某**PVC**。
- Released**: 绑定的**PVC**已经被删除，但资源尚未被集群回收。
- Failed**: 因自动回收资源失败而处于的故障状态。

7.5.2 创建PVC

PersistentVolumeClaim是存储卷类型的资源，它通过申请占用某个**PersistentVolume**而创建，它与**PV**是一一对应的关系，用户无须关心其底层实现细节。申请时，用户只需要指定目标空间的大小、访问模式、**PV**标签选择器和**StorageClass**等相关信息即可。**PVC**的**Spec**字段的可嵌套字段具体如下。

- accessMode**: 当前**PVC**的访问模式，其可用模式与**PV**相同。
- resources**: 当前**PVC**存储卷需要占用的资源量最小值；目前，**PVC**的资源限定仅指其空间大小。
- selector**: 绑定时对**PV**应用的标签选择器（**matchLabels**）或匹配条件表达式（**matchExpressions**），用于挑选要绑定的**PV**；如果同时指定了两种挑选机制，则必须同时满足两种选择机制的**PV**才能被选出。
- storageClassName**: 所依赖的存储类的名称。
- volumeMode**: 卷模型，用于指定此卷可被用作文件系统还是裸格式的块设备；默认为“**Filesystem**”。
- volumeName**: 用于直接指定要绑定的**PV**的卷名。

下面的配置清单（**pvc-nfs-0001.yaml**文件）定义了一个**PVC**资源示例，其选择**PV**的挑选机制是使用了标签选择器，适配的标签是**release: stable**，存储类为**slow**，这会关联到前面创建的**PV**资源**pv-nfs-0001**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs-0001
  labels:
    release: "stable"
spec:
  accessModes:
```



```
- ReadWriteMany
volumeMode: Filesystem
resources:
  requests:
    storage: 5Gi
storageClassName: slow
selector:
  matchLabels:
    release: "stable"
```

使用资源创建命令完成资源创建，而后即可查看其绑定PV资源的相关信息：

```
~]$ kubectl get pvc pvc-nfs-0001
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc-nfs-0001	Bound	pv-nfs-0001	5Gi	RWX	slow	6s

如果需要绑定此前创建的PV资源pv-rbd-0001，那么创建类似如下的资源配置即可，它将保存于配置文件pvc-rbd-0001.yaml中：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-rbd-0001
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 2Gi
  storageClassName: fast
  selector:
    matchLabels:
      release: "stable"
```

使用资源创建命令完成资源创建，而后即可查看其绑定PV资源的相关信息：

```
~]$ kubectl get pvc/pvc-rbd-0001
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc-rbd-0001	Bound	pv-rbd-0001	2Gi	RWO	fast	5s

创建好PVC资源之后，即可在Pod资源中通过persistentVolumeClaim引用它，而后挂载于容器中进行数据持久化。需要注意的是，

PV是集群级别的资源，而**PVC**则隶属于名称空间，因此，**PVC**在绑定目标**PV**时不受名称空间的限制，但**Pod**引用**PVC**时，则只能是属于同一名称空间中的资源。

7.5.3 在Pod中使用PVC

在Pod资源中调用PVC资源，只需要在定义volumes时使用persistentVolumeClaims字段嵌套指定两个字段即可，具体如下。

- claimName**: 要调用的PVC存储卷的名称，PVC卷要与Pod在同一名称空间中。

- readOnly**: 是否将存储卷强制挂载为只读模式，默认为false。

下面的清单定义了一个Pod资源，它是7.4.2节中直接使用RBD存储的Pod资源，此处改为调用了前面刚刚创建的名为pv-rbd-0001的PVC资源：

```
apiVersion: v1
kind: Pod
metadata:
  name: vol-rbd-pod
spec:
  containers:
    - name: redis
      image: redis:4-alpine
      ports:
        - containerPort: 6379
          name: redisport
      volumeMounts:
        - mountPath: /data
          name: redis-rbd-vol
      volumes:
        - name: redis-rbd-vol
          persistentVolumeClaim:
            claimName: pv-rbd-0001
```

资源创建完成后，即可通过类似于此前7.4.1节示例中的方式完成数据持久性测试。

7.5.4 存储类

存储类（**storage class**）是Kubernetes资源类型的一种，它是由管理员为管理PV之便而按需创建的类别（逻辑组），例如可按存储系统的性能高低分类，或者根据其综合服务质量级别进行分类（如图7-6所示）、依照备份策略分类，甚至直接按管理员自定义的标准进行分类等。不过，Kubernetes自身无法理解“类别”到底意味着什么，它仅仅是将这些当作PV的特性描述。

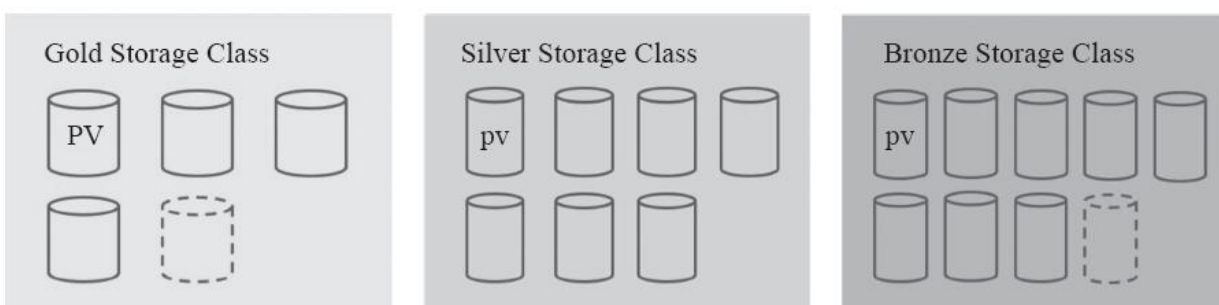


图7-6 基于综合服务质量的存储系统分类

存储类的好处之一便是支持PV的动态创建。用户用到持久性存储时，需要通过创建PVC来绑定匹配的PV，此类操作需求量较大，或者当管理员手动创建的PV无法满足PVC的所有需求时，系统按PVC的需求标准动态创建适配的PV会为存储管理带来极大的灵活性。

存储类对象的名称至关重要，它是用户调用的标识。创建存储类对象时，除了名称之外，还需要为其定义三个关键字段：**provisioner**、**parameter**和**reclaimPolicy**。

1.StorageClass Spec

StorageClass Spec中的字段是定义存储类时最重要的字段，其包含以下五个可用字段。

- provisioner**（供给方）：即提供了存储资源的存储系统，存储类要依赖Provisioner来判定要使用的存储插件以便适配到目标存储系统。Kubernetes内建有多种供给方（**Provisioner**），这些供给方的名字都

以“kubernetes.io”为前缀。另外，它还支持用户依据Kubernetes规范自定义Provisioner。

- parameters（参数）：存储类使用参数描述要关联到的存储卷，不过，不同的Provisioner可用的参数各不相同。

- reclaimPolicy：为当前存储类动态创建的PV指定回收策略，可用值为Delete（默认）和Retain；不过，那些由管理员手工创建的PV的回收策略则取决于它们自身的定义。

- volumeBindingMode：定义如何为PVC完成供给和绑定，默认值为“VolumeBinding Immediate”；此选项仅在启用了存储卷调度功能时才能生效。

- mountOptions：由当前类动态创建的PV的挂载选项列表。

下面是一个定义在glusterfs-storageclass.yaml配置文件中的资源清单，它定义了一个使用Gluster存储系统的存储类glusterfs，并通过annotations字段将其定义为默认的存储类：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: glusterfs
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://heketi.ilinux.io:8080"
  restauthenabled: "false"
  restuser: "ik8s"
  restuserkey: "ik8s.io"
```

这里需要特别提醒读者的是，parameters.resturl字段用于指定Gluster存储系统的RESTful风格的访问接口，本示例中使用的“<http://heketi.ilinux.io:8080>”应替换为读者自己实际环境中的可用地址。Gluster存储系统本身并不支持这种访问方式，管理员需要额外部署heketi配合Gluster以提供此类服务接口。Heketi支持认证访问，不过只有在restauthenabled设置为“true”时，restuser和restuserkey字段才会启用。Heketi的设置及使用方式请参考附录B。

2.动态PV供给

动态PV供给的启用，需要事先由管理员创建至少一个存储类，不同的Provisioner的创建方法各有不同，具体内容如前一节所示。另外，并非所有的存储卷插件都由Kubernetes内建支持PV动态供给功能，具体信息如图7-7所示。

上文中定义glusterfs存储类资源创建完成后，便可以据此使用动态PV供给功能。下面的资源清单定义在pvc-gluserfs-dynamic-0001.yaml配置文件中，它将从glusterfs存储类中申请使用5GB的存储空间：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-gluster-dynamic-0001
  annotations:
    volume.beta.kubernetes.io/storage-class: glusterfs
spec:
  # storageClassName: "glusterfs"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Volume Plugin	Internal Provisioner
AWSElasticBlockStore	✓
AzureFile	✓
AzureDisk	✓
CephFS	-
Cinder	✓
FC	-
FlexVolume	-
Flocker	✓
GCEPersistentDisk	✓
Glusterfs	✓
iSCSI	-
PhotonPersistentDisk	✓
Quobyte	✓
NFS	-
RBD	✓
VsphereVolume	✓
PortworxVolume	✓
ScaleIO	✓
StorageOS	✓
Local	-

图7-7 各存储插件对动态供给方式的支持状况

目前，在PVC的定义中指定使用的存储类资源的方式共有两种：一种是使用spec.storageClassName字段，另一种是使用“volume.beta.kubernetes.io/storage-class”注解信息，如上面示例中所示。不过，建议仅使用一种方式，以免两者设置为不同的值时会出现配置错误。接下来创建定义的PVC，并检查其绑定状态：

```
~]$ kubectl create -f pvc-glusterfs-dynamic-0001.yaml
persistentvolumeclaim "pvc-gluster-dynamic-0001" created
```

通过如下命令输出的PVC资源的描述信息可以看到，PVC存储卷已创建完成且已经完成了PV绑定，绑定的PV资源由persistentvolume-controller控制器动态提供：

```
~]$ kubectl describe pvc pvc-gluster-dynamic-0001
Name:          pvc-gluster-dynamic-0001
Namespace:     default
StorageClass:  glusterfs
Status:        Bound
Volume:        pvc-5836eb47-6c77-11e8-9bab-000c29be4e28
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed=yes
               pv.kubernetes.io/bound-by-controller=yes
               volume.beta.kubernetes.io/storage-
provisioner=kubernetes.io/glusterfs
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      5Gi
Access Modes:  RWO
Events:
  Type      Reason              Age   From                      Message
  ----      -
  Normal    ProvisioningSucceeded 55s   persistentvolume-controller Successfully
provisioned volume pvc-5836eb47-6c77-11e8-9bab-000c29be4e28 using
kubernetes.
io/glusterfs
```

任何支持PV动态供给的存储系统都可以在定义为存储类后由PVC动态申请使用，这对于难以事先预估使用到的存储空间大小及存储卷数量的使用场景尤为有用，例如由StatefulSet控制器管理Pod对象时，存储卷是必备资源，且随着规模的变动，存储卷的数量也会随之变动。

另外，用户也可以使用云端存储提供的PV动态供给机制，如AWS EBS、AzureDisk、Cinder或GCEPersistentDisk等，将Kubernetes部署于IaaS云端时，此种存储方式使用的较多。各类云存储动态供给的具体使用方式请参考相关的使用手册。

7.5.5 PV和PVC的生命周期

PV是Kubernetes集群的存储资源，而PVC则代表着资源需求。创建PVC时对PV发起的使用申请，即为“绑定”。PV和PVC是一一对应的关系，可用于响应PVC申请的PV必须要能够容纳PVC的请求条件，它们二者的交互遵循如下生命周期。

1. 存储供给

存储供给（Provisioning）是指为PVC准备可用PV的机制。Kubernetes支持两种PV供给方式：静态供给和动态供给。

（1）静态供给

静态供给是指由集群管理员手动创建一定数量的PV的资源供应方式。这些PV负责处理存储系统的细节，并将其抽象成易用的存储资源供用户使用。静态提供的PV可能属于某存储类（StorageClass），也可能没有存储类，这一点取决于管理员的设定。

（2）动态供给

不存在某静态的PV匹配到用户的PVC申请时，Kubernetes集群会尝试为PVC动态创建符合需求的PV，此即为动态供给。这种方式依赖于存储类的辅助，PVC必须向一个事先存在的存储类发起动态分配PV的请求，没有指定存储类的PVC请求会被禁止使用动态创建PV的方式。

另外，为了支持使用动态供给机制，集群管理员需要为准入控制器（admission controller）启用“DefaultStorageClass”选项，这一点通过“--admission-control”命令行选项为API Server进行设定即可，后文会对准入控制器予以描述。

2. 存储绑定

用户基于一系列存储需求和访问模式定义好PVC后，Kubernetes系统的控制器即会为其查找匹配的PV，并于找到之后在此二者之间建立

起关联关系，而后它们二者之间的状态即转为“绑定”（Binding）。若PV是为PVC而动态创建的，则该PV专用于其PVC。

若是无法为PVC找到可匹配的PV，则PVC将一直处于未绑定（unbound）状态，直到有符合条件的PV出现并完成绑定方才可用。

（1）存储使用（Using）

Pod资源基于persistentVolumeClaim卷类型的定义，将选定的PVC关联为存储卷，而后即可为内部的容器所使用。对于支持多种访问模式的存储卷来说，用户需要额外指定要使用的模式。一旦完成将存储卷挂载至Pod对象内的容器中，其应用即可使用关联的PV提供的存储空间。

（2）PVC保护（Protection）

为了避免使用中的存储卷被移除而导致数据丢失，Kubernetes自1.9版本起引入了“PVC保护机制”。启用了此特性后，万一有用户删除了仍处于某Pod资源使用中的PVC时，Kubernetes不会立即予以移除，而是推迟到不再被任何Pod资源使用后才执行删除操作。处于此种阶段的PVC资源的status字段为“Termination”，并且其Finalizers字段中包含“kubernetes.io/pvc-protection”。

3.存储回收（Reclaiming）

完成存储卷的使用目标之后，即可删除PVC对象以便进行资源回收。不过，至于如何操作则取决于PV的回收策略。目前，可用的回收策略有三种：Retained、Recycled和Deleted。

（1）留存（Retain）

留存策略意味着在删除PVC之后，Kubernetes系统不会自动删除PV，而仅仅是将它置于“释放”（released）状态。不过，此种状态的PV尚且不能被其他PVC申请所绑定，因为此前的申请生成的数据仍然存在，需要由管理员手动决定其后续处理方案。这就意味着，如果想要再次使用此类的PV资源，则需要由管理员按下面的步骤手动执行删除操作。

- 1) 删除PV，这之后，此PV的数据依然留存于外部的存储之上。
- 2) 手工清理存储系统上依然留存的数据。
- 3) 手工删除存储系统级的存储卷（例如，RBD存储系统上的image）以释放空间，以便再次创建，或者直接将其重新创建为PV。

（2）回收（Recycle）

如果可被底层存储插件支持，资源回收策略会在存储卷上执行数据删除操作并让PV资源再次变为可被Claim。另外，管理员也可以配置一个自定义的回收器Pod模板，以便执行自定义的回收操作。不过，此种回收策略行将废弃。

（3）删除（Delete）

对于支持Deleted回收策略的存储插件来说，在PVC被删除后会直接移除PV对象，同时移除的还有PV相关的外部存储系统上的存储资产（asset）。支持这种操作的存储系统有AWS EBS、GCE PD、Azure Disk或Cinder。动态创建的PV资源的回收策略取决于相关存储类上的定义，存储类上相关的默认策略为Delete，大多数情况下，管理员都需要按用户期望的处理机制修改此默认策略，以免导致数据非计划内的误删除。

4.扩展PVC

Kubernetes自1.8版本起增加了扩展PV空间的特性，截至目前，它所支持的扩展PVC机制的存储卷共有以下几种。

- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd

“PersistentVolumeClaimResize”准入插件负责对支持空间大小变动的存储卷执行更多的验证操作，管理员需要事先启用此插件才能使用PVC扩展机制，那些将“allowVolume Expansion”字段的值设置为“true”的存储类即可动态扩展存储卷空间。随后，用户改动Claim请求更大的空间即能触发底层PV空间扩展从而带来PVC存储卷的扩展。

对于包含文件系统的存储卷来说，只有在有新的Pod资源基于读写模式开始使用PVC时才会执行文件系统的大小调整操作。换句话说，如果某被扩展的存储卷已经由Pod资源所使用，则需要重建此Pod对象才能触发文件系统大小的调整操作。支持空间调整的文件系统仅有XFS和EXT3/EXT4。

7.6 downwardAPI存储卷

很多时候，应用程序需要基于其所在的环境信息设定运行特性等，这类环境信息包括节点及集群的部分详细属性信息等，例如，Nginx进程可根据节点的CPU核心数量自动设定要启动的worker进程数，JVM虚拟机可根据节点内存资源自动设定其堆内存大小。类似地，托管运行于Kubernetes的Pod对象中的容器化应用偶尔也需要获取其所属Pod对象的IP、主机名、标签、注解、UID、请求的CPU及内存资源量及其限额，甚至是Pod所在的节点名称等，容器可以通过环境变量或downwardAPI存储卷访问此类信息，不过，标签和注解仅支持通过存储卷暴露给容器。

7.6.1 环境变量式元数据注入

引用downwardAPI元数据信息的常用方式之一是使用容器的环境变量，它通过在valueFrom字段中嵌套fieldRef或resourceFieldRef字段来引用相应的数据源。不过，通常只有常量类的属性才能够通过环境变量注入到容器中，毕竟，在进程启动完成后将无法再向其告知变量值的变动，于是，环境变量也就不支持中途的更新操作。

可通过fieldRef字段引用的信息具体如下。

- spec.nodeName: 节点名称。
- status.hostIP: 节点IP地址。
- metadata.name: Pod对象的名称。
- metadata.namespace: Pod对象隶属的名称空间。
- status.podIP: Pod对象的IP地址。
- spec.serviceAccountName: Pod对象使用的ServiceAccount资源的名称。
- metadata.uid: Pod对象的UID。
- metadata.labels['<KEY>']: Pod对象标签中的指定键的值，例如metadata.labels['mylabel']，仅Kubernetes 1.9及之后的版本才支持。
- metadata.annotations['<KEY>']: Pod对象注解信息中的指定键的值，仅Kubernetes 1.9及之后的版本才支持。

另外，可通过resourceFieldRef字段引用的信息是指当前容器的资源请求及资源限额的定义，因此它们包括requests.cpu、limits.cpu、requests.memory和limits.memory四项。

下面的资源配置清单示例（downwardAPI-env.yaml）中定义的Pod对象通过环境变量向容器env-test-container中注入了Pod对象的名称、

隶属的名称空间、标签app的值以及容器自身的CPU资源限额和内存资源请求等信息：

```
apiVersion: v1
kind: Pod
metadata:
  name: env-test-pod
  labels:
    app: env-test-pod
spec:
  containers:
    - name: env-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
  env:
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_APP_LABEL
      valueFrom:
        fieldRef:
          fieldPath: metadata.labels['app']
    - name: MY_CPU_LIMIT
      valueFrom:
        resourceFieldRef:
          resource: limits.cpu
    - name: MY_MEM_REQUEST
      valueFrom:
        resourceFieldRef:
          resource: requests.memory
          divisor: 1Mi
  restartPolicy: Never
```

此Pod对象创建完成后，向控制台打印所有的环境变量即可终止运行，它仅用于测试通过环境变量注入信息到容器的使用效果：

```
~]$ kubectl create -f downwardAPI-env.yaml
pod "env-test-pod" created
$ kubectl get pods -l app=env-test-pod
NAME          READY    STATUS    RESTARTS   AGE
env-test-pod  0/1      Completed 0           1m
```

而后即可通过控制台日志获取注入的环境变量：

```
~]$ kubectl logs env-test-pod | grep "^MY_"
MY_POD_NAMESPACE=default
MY_CPU_LIMIT=1
MY_APP_LABEL=env-test-pod
MY_MEM_REQUEST=32
MY_POD_NAME=env-test-pod
```

如示例中的最后一个环境变量所示，在定义资源请求或资源限制时还可额外指定一个“**divisor**”字段，用于为引用的值指定一个除数以实现所引用的相关值的单位换算。CPU资源的**divisor**字段其默认值为1，表示为1个核心，相除的结果不足1个单位时则向上圆整（例如，0.25向上圆整的结果为1），它的另一个可用单位为1m，即表示1个微核心。内存资源的**divisor**字段其默认值也是1，不过，它意指1个字节，此时，32Mi的内存资源则要换算为33554432的结果予以输出。其他可用的单位还有1Ki、1Mi、1Gi等，于是，在将**divisor**字段的值设置为1Mi时，32Mi的内存资源的换算结果即为32。



注意 未为容器定义资源请求及资源限额时，downwardAPI引用的值即默认为节点的可分配CPU及内存资源量。

7.6.2 存储卷式元数据注入

向容器注入元数据信息的另一种方式是使用downwardAPI存储卷，它将配置的字段数据映射为文件并可通过容器中的挂载点进行访问。7.2.5节中能够通过环境变量的方式注入的元数据信息也都可以使用存储卷的方式进行信息暴露，除此之外，还可以在downwardAPI存储卷中使用fieldRef引用如下两个数据源。

- `metadata.labels`: Pod对象的所有标签信息，每行一个，格式为`label-key="escaped-label-value"`。

- `metadata.annotations`: Pod对象的所有注解信息，每行一个，格式为`annotation-key="escaped-annotation-value"`。

下面的资源配置清单示例（`downwardAPI-vol.yaml`）中定义的Pod对象通过downwardAPI存储卷向容器`volume-test-container`中注入了Pod对象隶属的名称空间、标签、注解以及容器自身的CPU资源限额和内存资源请求等信息。存储卷在容器中的挂载点为`/etc/podinfo`目录，因此，注入的每一项信息均会映射为此路径下的一个文件：

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: east-china
    rack: rack-101
    app: dapi-vol-pod
  name: dapi-vol-pod
  annotations:
    annotation1: "test-value-1"
spec:
  containers:
    - name: volume-test-container
      image: busybox
      command: ["sh", "-c", "sleep 864000"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
```

```
    readOnly: false
volumes:
- name: podinfo
  downwardAPI:
    defaultMode: 420
    items:
    - fieldRef:
        fieldPath: metadata.namespace
        path: pod_namespace
    - fieldRef:
        fieldPath: metadata.labels
        path: pod_labels
    - fieldRef:
        fieldPath: metadata.annotations
        path: pod_annotations
    - resourceFieldRef:
        containerName: volume-test-container
        resource: limits.cpu
        path: "cpu_limit"
    - resourceFieldRef:
        containerName: volume-test-container
        resource: requests.memory
        divisor: "1Mi"
        path: "mem_request"
```

创建资源配置清单中定义的Pod对象后即可测试访问由downwardAPI存储卷映射的文件pod_namespace、pod_labels、pod_annotations、limits_cpu和mem_request等：

```
~]$ kubectl create -f downwardAPI-vol.yaml
pod "dapi-vol-pod" created
```

接下来即可测试访问上述的映射文件，例如，查看Pod对象的标签列表：

```
~]$ kubectl exec dapi-vol-pod -- cat /etc/podinfo/pod_labels
app="dapi-vol-pod"
rack="rack-101"
zone="east-china"
```

如命令结果所示，Pod对象的标签信息每行一个地映射于自定义的路径/etc/podinfo/pod_labels文件中，类似地，注解信息也以这种方式进行处理。如前面的章节中所述，标签和注解支持运行时修改，其改动的结果也会实时映射进downwardAPI生成的文件中。例如，为dapi-vol-pod添加新的标签：

```
~]$ kubectl label pods dapi-vol-pod env="test"  
pod "dapi-vol-pod" labeled
```

而后再次查看容器内的`pod_labels`文件的内容，由如下的命令结果可知新的标签已经能够通过相关的文件获取到：

```
~]$ kubectl exec dapi-vol-pod -- cat /etc/podinfo/pod_labels  
app="dapi-vol-pod"  
env="test"  
rack="rack-101"  
zone="east-china"
```

downwardAPI存储卷为Kubernetes上运行容器化应用提供了获取外部环境信息的有效途径，这一点对那些非为云原生开发的应用程序在不进行代码重构的前提下，获取环境信息进行自身配置等操作时尤为有用。

7.7 本章小结

本章主要讲解了Kubernetes的存储卷及其功用，并通过应用示例给出了部署存储卷类型的使用方法，具体如下。

- 临时存储卷emptyDir和gitRepo的生命周期与Pod对象相同，但gitRepo能够通过引用外部Git仓库的数据来实现数据的持久性。

- 节点存储卷hostPath提供了节点级别的数据持久能力。

- 网络存储卷NFS、GlusterFS和RBD等是企业内部较为常用的独立部署的持久存储系统。

- 云存储卷AWS ebs等是托管于云端的Kubernetes系统上较为常用的持久存储系统。

- PV和PVC可将存储管理和存储使用解耦为消费者模型。

- 基于StorageClass可以实现PV的动态供给，GlusterFS和Ceph RBD，以及云端存储AWS ebs等都可以实现此类功能。

第8章 配置容器应用：ConfigMap和Secret

ConfigMap和Secret是Kubernetes系统上两种特殊类型的存储卷，ConfigMap对象用于为容器中的应用提供配置数据以定制程序的行为，不过敏感的配置信息，例如密钥、证书等通常由Secret对象来进行配置。它们将相应的配置信息保存于对象中，而后在Pod资源上以存储卷的形式将其挂载并获取相关的配置，以实现配置与镜像文件的解耦。本章将主要讲解ConfigMap与Secret存储卷的用法。

8.1 容器化应用配置方式

每个应用程序都是一个可执行程序文件，它包含操作码列表，CPU通过执行这些操作码来完成特定的操作。例如，`cat`命令是由`/usr/bin/cat`文件提供的，该文件含有机指令的列表，在屏幕上显示指定文件的内容时需要使用这些机器指令。几乎每个程序的行为都可以通过其命令行选项及参数或配置文件来按需定制。实践中，人们通常不会以默认的配置参数运行应用程序，而是需要根据特定的环境或具体的需求定制其运行特性，对于复杂的服务类应用程序更是如此，如Nginx、Tomcat和HBase等，而且通过配置文件定义其配置通常是首选甚至是唯一的途径。

那么，如何为容器中的应用提供配置信息呢？例如，为Nginx配置一个特定Server或指定worker进程的数量，为Tomcat的JVM配置其堆内存的大小等。传统实践中，通常有这么几种途径：启动容器时直接向命令传递参数、将定义好的配置文件硬编码于（嵌入）镜像文件中、通过环境变量（Environment Variables）传递配置数据，以及基于Docker卷传送配置文件等。

1.通过命令行参数进行配置

Docker容器可用来运行单个应用程序。在制作Docker镜像时，Dockerfile中的ENTRYPOINT和CMD指令可用于指定容器启动时要运行的程序及其相关的参数。CMD指令以列表的形式指定要运行的程序及其相关的参数，但若同时存在ENTRYPOINT指令，则CMD指令中列表的所有元素均将被视作是由ENTRYPOINT指定的程序的命令行参数。另外，在基于某镜像使用Docker命令创建容器时，可以在命令行向ENTRYPOINT中的程序传递额外的自定义参数，甚至还可以修改要运行的应用程序本身。例如，使用`docker run`命令创建并启动容器的格式为：

```
docker run[OPTIONS]IMAGE[COMMAND][ARG...]
```

其中的[COMMAND]即为自定义运行的程序，[ARG]则是传递给程序的参数。若定义相关的镜像文件时使用了ENTRYPOINT指令，则

[COMMAND]和[ARG]都会被当作命令行参数传递给ENTRYPOINT指令中指定的程序，除非为docker run命令额外使用--entrypoint选项覆盖ENTRYPOINT而指定运行其他程序。相关的使用详情请参考Docker的相关教程。

在Kubernetes系统上创建Pod资源时，也能够向容器化应用传递命令行参数，甚至指定运行其他应用程序，相关的字段分别为pods.spec.containers.command和pods.spec.containers.args。这一点在前面相关的章节中已有相关的使用说明。

2.将配置文件嵌入镜像文件

所谓的将配置文件嵌入镜像文件，是指用户在Dockerfile中使用COPY指令把定义好的配置文件复制到镜像文件系统上的目标位置，或者使用RUN指令调用sed或echo一类的命令修改配置文件从而达到为容器化应用提供自定义配置文件之目的。使用时，若Docker Hub上的某镜像文件额外添加配置文件即能符合需要，则克隆其Dockerfile文件修改至符合需求之后再将其推送至GitHub，并由Docker Hub自动构建出镜像文件即可。

这种方式的优势在于对于用户来说简单易用，不用任何额外的设定就能启动符合需求的容器，用于Kubernetes环境亦无须多余的配置。但配置文件相关的任何额外的修改需求都不得不通过重新构建镜像文件来实现，路径长、效率低。

3.通过环境变量向容器注入配置信息

通过环境变量为镜像提供配置信息是Docker Hub上最常见的使用方式。例如，使用MySQL官方提供的镜像文件启动MySQL容器时使用的MYSQL_ROOT_PASSWORD环境变量，它用于为MySQL服务器的root用户设置登录密码。

在基于此类镜像启动容器时，用户为docker run命令通过-e选项向环境变量传值即能实现应用配置，命令的使用格式为“docker run-e SETTING1=foo-e SETTING2=bar...<image name>”。启动时，容器的ENTRYPOINT启动脚本会抓取到这些环境变量，并在启动容器应用之前，通过sed或echo等一类的命令将变量值替换到配置文件中。

一般说来，容器的ENTRYPOINT脚本应该为这些环境变量提供默认值，以便在用户未为环境变量传值时也能基于此类需要环境变量的镜像启动容器。使用环境变量这种配置方式的优势在于配置信息的动态化供给，不过，有些应用程序的配置可能会复杂到无法通过key/value格式的环境变量完成。

另外，也可以让容器的ENTRYPOINT启动脚本通过网络中的K/V存储获取配置参数，常用的此类存储系统有Consul或etcd等。这种方式较之简单的环境变量能够提供更复杂的配置信息，因为KV存储系统支持多层级的嵌套数据结构，有一些应用广泛的数据抓取工具能够从KV存储中加载相关的数据并替换于配置文件中，甚至于像confd这类的工具还能在KV中的数据变化时自动将其重载至配置文件中，这一点实现了真正意义上的配置动态化。不过，这种方式为容器化应用引入了额外的依赖条件。

Kubernetes系统支持在为Pod资源配置容器时使用spec.containers.env为容器的环境变量传值从而完成应用的配置。如前所述，这种方式无法应付较复杂的配置场景，因此提供的配置能力也很有限。

4.通过存储卷向容器注入配置信息

Docker存储卷（volumes）能够将宿主机之上的任何文件或目录映射到容器文件系统上，因此，可以事先将配置文件放置于宿主机之上的某特定路径中，而后在启动容器时进行加载。这种方式灵活易用，但也依赖于用户需要事先将配置数据提供在宿主机上的特定路径下，而且在多主机模型中，若容器存在被调度至任一主机运行的可能性时，用户还需要将配置共享到任一宿主机以确保容器能够正确地获取到它们。

5.借助Docker config进行容器配置

Docker swarm service自1.13版本起支持使用secret于容器之外保存二进制数据，包括口令、SSH私钥、SSL证书以及其他不建议通过网络传输或不应该在Dockerfile及程序源码中非加密保存的机密数据。用户可使用secret集中化管理这类数据并将其安全关联至那些需要访问这些数据的容器中。

另外，**Docker**自17.06版本起为**swarm service**引入了允许用户于容器之外存储非敏感信息（如配置文件）的组件“**service config**”，从而支持用户创建通用目的的镜像文件，并且不再需要通过挂载存储卷或使用环境变量为容器提供配置文件。

Docker swarm service secret和**config**为容器化应用的配置提供了极大的灵活性，不过，它们也只能应用于**Docker swarm service**环境中，而不能应用于单独运行的容器之上。

Kubernetes系统也有类似的组件，它们称为**Secret**和**ConfigMap**，而且是**Kubernetes**系统上一等类别的资源对象，它们要么被**Pod**资源以存储卷的形式加载，要么由容器通过**envFrom**字段以变量的形式加载。

8.2 通过命令行参数配置容器应用

创建Pod资源时，可以在容器定义中自定义要运行的命令以及为其传递的选项和参数。在容器的配置上下文中，使用`command`字段指定要运行的程序，而`args`字段则可用于指定传递给程序的选项及参数。在配置文件中定义的`command`和`args`会覆盖镜像文件中相关的默认设定，这类程序会被直接运行，而不会由`shell`解释器解释运行，因此与`shell`相关的特性均不被支持，如命令行展开符号`{}`、重定向等操作。

下面是定义在`command-demo.yaml`文件中的Pod资源示例，它在容器`command-demo-container`中将`busybox`镜像文件中默认运行的命令`["/bin/sh", "-c"]`修改为`["httpd"]`，并为其额外传递了`["-f"]`选项：

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
    - name: command-demo-container
      image: busybox
      command: ["httpd"]
      args: ["-f"]
      ports:
        - containerPort: 80
      restartPolicy: OnFailure
```

事实上，用户也可以只在容器配置的上下文中提供`args`字段，以实现向默认运行的程序提供额外的参数。如果默认的命令为`Shell`解释器或`entrypoint`启动脚本，那么这些参数本身甚至还可以是要运行的命令及其参数。例如，下面的容器配置，表示要运行的程序为`“/bin/sh-c httpd-f”`，实现了以`shell`解释器解释运行指定的程序之目的：

```
spec:
  containers:
    - name: command-demo-container
      image: busybox
      args: ["httpd", "-f"]
      ports:
        - containerPort: 80
```

由上述的应用示例可见，Kubernetes配置文件中的command对应于Dockerfile中的ENTRYPOINT，而配置文件中的args则对应于Dockerfile中的CMD。在Kubernetes中只给出command字段时，它会覆盖Dockerfile中的ENTRYPOINT和CMD，只给出args字段时，它仅覆盖CMD，而同时给出command和args时，会对应覆盖ENTRYPOINT和CMD。其应用生效的示例如图8-1所示。

通过命令行参数的方式向容器应用传递配置数据的操作比较简单，但其功能有限，难以为应用生成复杂配置，尤其是那些不支持通过命令行参数进行的配置将无法基于这种机制来实现。另外需要注意的是，在容器创建完成后，修改command和args并不会直接生效，除非重建Pod对象。

Image Entrypoint	Image Cmd	Container command	Container args	Command run
[/ep-1]	[foo bar]	<not set>	<not set>	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	<not set>	[ep-2]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	[/ep-2]	[zoo boo]	[ep-2 zoo boo]

图8-1 command/args和ENTRYPOINT/CMD

8.3 利用环境变量配置容器应用

于运行时配置Docker容器中应用程序的第二种方式是在容器启动时向其传递环境变量。Docker原生的应用程序应该使用很小的配置文件，并且每一项参数都可由环境变量或命令行选项覆盖，从而能够在运行时完成任意的按需配置。然而，目前只有极少一部分应用程序为容器环境原生设计，毕竟为容器原生重构应用程序工程浩大，且旷日持久。好在有通过容器启动脚本为应用程序预设运行环境的方法可用，通行的做法是在制作Docker镜像时，为ENTRYPOINT指令定义一个脚本，它能够在启动容器时将环境变量替换至应用程序的配置文件中，而后再由此脚本启动相应的应用程序。基于这类镜像运行容器时，即可通过向环境变量传值的方式来配置应用程序。

在Kubernetes中使用此类镜像启动容器时，也可以在Pod资源或Pod模板资源的定义中，为容器配置段使用env参数来定义所使用的环境变量列表。事实上，即便容器中的应用本身不处理环境变量，也一样可以向容器传递环境变量，只不过它不被使用罢了。

环境变量配置容器化应用时，需要在容器配置段中嵌套使用env字段，它的值是一个由环境变量构建的列表。环境变量通常由name和value（或valueFrom）字段构成。

·name<string>：环境变量的名称，必选字段。

·value<string>：环境变量的值，通过\$(VAR_NAME)引用，逃逸格式为“\$\$ (VAR_NAME)”，默认值为空。

·valueFrom<Object>：环境变量值的引用源，例如，当前Pod资源的名称、名称空间、标签等，不能与非空值的value字段同时使用，即环境变量的值要么源于value字段，要么源于valueFrom字段，二者不可同时提供数据。

valueFrom字段可引用的值有多种来源，包括当前Pod资源的属性值，容器相关的系统资源配置、ConfigMap对象中的Key以及Secret对象中的Key，它们应分别使用不同的嵌套字段进行定义。

·fieldRef<Object>：当前Pod资源的指定字段，目前支持使用的字段包括metadata.name、metadata.namespace、metadata.labels、metadata.annotations、spec.nodeName、spec.serviceAccountName、status.hostIP和status.podIP。

·configMapKeyRef<Object>：ConfigMap对象中的特定Key。

·secretKeyRef<Object>：Secret对象中的特定Key。

·resourceFieldRef<Object>：当前容器的特定系统资源的最小值（配额）或最大值（限额），目前支持的引用包括limits.cpu、limits.memory、limits.ephemeral-storage、requests.cpu、requests.memory和requests.ephemeral-storage。

下面是定义在配置文件env-demo.yaml中的Pod资源，其通过环境变量引用当前Pod资源及其所在的节点的相关属性值配置容器。fieldRef字段的值是一个对象，它一般由apiVersion（创建当前Pod资源的API版本）或fieldPath嵌套字段所定义：

```
apiVersion: v1
kind: Pod
metadata:
  name: env-demo
  labels:
    purpose: demonstrate-environment-variables
spec:
  containers:
    - name: env-demo-container
      image: busybox
      command: ["httpd"]
      args: ["-f"]
      env:
        - name: HELLO_WORLD
          value: just a demo
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_NODE_IP
          valueFrom:
            fieldRef:
              fieldPath: status.hostIP
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      restartPolicy: OnFailure
```

创建上面资源清单中定义的Pod对象env-demo，而后打印它的环境变量列表、命令及其结果如下：

```
~]$ kubectl exec env-demo printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=env-demo
MY_NODE_NAME=node02.ilinux.io
MY_NODE_IP=172.16.0.67
MY_POD_NAMESPACE=default
HELLO_WORLD=just a demo
.....
```

容器的启动脚本或应用程序调用或处理这些环境变量，即可实现容器化应用的配置。相较于命令行参数的方式来说，使用环境变量的配置方式更清晰、易懂，尤其是对于首次使用相关容器的用户来说，这种方式能够快速了解容器的配置方式。不过，这两种配置方式有一个共同的缺陷：无法在容器应用运行过程中更新环境变量从而达到更新应用之目的。这通常意味着用户不得不为production、development和qa等不同的环境分别配置Pod资源。好在，用户还有ConfigMap资源可用。

8.4 应用程序配置管理及ConfigMap资源

分布式环境中，基于负载、容错等需求的考虑，几乎所有的服务都需要在不同的机器节点上部署不止一个实例。随着程序功能的日益复杂，程序的配置日益增多，而且配置文件的修改频率通常远远大于代码本身，这种情况下，有时仅仅是一个配置内容的修改，就不得不重新进行代码提交到SVN/Git、打包、分发上线的流程。部署规则较大的场景中，分发上线工作既繁杂又沉重。

究其根本，所有的这些麻烦都是由于配置和代码在管理和发布过程中不加区分所致。配置本身源于代码，是为了提高代码的灵活性而提取出来的一些经常变化的或需要定制的内容，而正是配置的这种天生的变化特征为部署过程带来了不小的麻烦，也最终催生了分布式系统配置管理系统，将配置内容从代码中完全分离出来，及时可靠高效地提供配置访问和更新服务。



提示 国内分布式配置中心相关的开源项目有Diamond（阿里）、Apollo（携程）、Qconf（奇虎360）和disconf（百度）等。

作为分布式系统的Kubernetes也提供了统一配置管理方案——ConfigMap。Kubernetes基于ConfigMap对象实现了将配置文件从容器镜像中解耦，从而增强了容器应用的可移植性。简单来说，一个ConfigMap对象就是一系列配置数据的集合，这些数据可“注入”到Pod对象中，并为容器应用所使用，注入方式有挂载为存储卷和传递为环境变量两种。

ConfigMap对象将配置数据以键值对的形式进行存储，这些数据可以在Pod对象中使用或者为系统组件提供配置，例如控制器对象等。不过，无论应用程序如何使用ConfigMap对象中的数据，用户都完全可以通过在不同的环境中创建名称相同但内容不同的ConfigMap对象，从而为不同环境中同一功能的Pod资源提供不同的配置信息，实现应用与配置的灵活勾兑。

8.4.1 创建ConfigMap对象

Kubernetes的不少资源既可以使用`kubectl create`命令创建，也可以使用清单创建，例如前面讲到的`namespace`。ConfigMap是另一个两种创建方式都比较常用的资源。而且，通过使用“`kubectl create configmap`”命令，用户可以根据目录、文件或直接值创建ConfigMap对象。命令的语法格式如下所示：

```
kubectl create configmap<map-name><data-source>
```

其中，`<map-name>`即为ConfigMap对象的名称，而`<data-source>`是数据源，它可以通过直接值、文件或目录来获取。无论是哪一种数据源供给方式，它都要转换为ConfigMap对象中的Key-Value数据，其中Key由用户在命令行给出或是文件数据源的文件名，它仅能由字母、数字、连接号和点号组成，而Value则是直接值或文件数据源的内容。

1.利用直接值创建

为“`kubectl create configmap`”命令使用“`--from-literal`”选项可在命令行直接给出键值对来创建ConfigMap对象，重复使用此选项则可以传递多个键值对。命令格式如下：

```
kubectl create configmap configmap_name --from-literal=key-name-1=value-1
```

例如，下面的命令创建`special-config`时传递了两个键值对：

```
~]$ kubectl create configmap special-config \
--from-literal=special.how=very --from-literal=special.type=charm
```

“`get configmap`”命令可用于查看创建的ConfigMap对象`special-config`的相关信息，例如，如下的命令及其结果：

```
$ kubectl get configmaps special-config -o yaml
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2018-03-17T05:24:56Z
  name: special-config
  namespace: default
  resourceVersion: "465543"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: 87b1bda2-29a3-11e8-b246-000c29be4e28
```

此类方式提供的数据量有限，一般是在仅通过有限的几个数据项即可为Pod资源提供足够的配置信息时使用。

2. 基于文件创建

为“`kubectl create configmap`”命令使用“`--from-file`”选项即可基于文件内容来创建ConfigMap对象，它的命令格式如下。可以重复多次使用“`--from-file`”选项以传递多个文件内容：

```
kubectl create configmap <configmap_name> --from-file=<path-to-file>
```

例如，下面的命令可以把事先准备好的Nginx配置文件模板保存于ConfigMap对象nginx-config中：

```
~]$ kubectl create configmap nginx-config \
--from-file=./data/configs/nginx/conf.d/myserver.conf
```

这种方式创建的ConfigMap对象，其数据存储的键为文件路径的基名，值为文件内容，例如下面命令显示的nginx-config对象的信息：

```
~]$ kubectl get configmap nginx-config -o yaml
apiVersion: v1
data:
  myserver.conf: |
    server {
      listen 8080;
      server_name www.ilinux.io;

      include /etc/nginx/conf.d/myserver-*.cfg;

      location / {
```

```
        root /usr/share/nginx/html;
    }
}
kind: ConfigMap
.....
```

如果需要自行指定键名，则可在“**--from-file**”选项中直接指定自定义的键，命令格式如下：

```
kubectl create configmap <configmap_name> --from-file= <my-key-name>=<path-to-file>
```

通过这种方式创建的**ConfigMap**资源可以直接以键值形式收纳应用程序的完整配置信息，多个文件可分别存储于不同的键值当中。另外需要说明的是，基于直接值和基于文件创建的方式也可以混编使用。

3.基于目录创建

如果配置文件数量较多且存储于有限的目录中时，**kubectl**还提供了基于目录直接将多个文件分别收纳为键值数据的**ConfigMap**资源创建方式。将“**--from-file**”选项后面所跟的路径指向一个目录路径就能将目录下的所有文件一同创建于同一**ConfigMap**资源中，命令格式如下：

```
kubectl create configmap <configmap_name> --from-file=<path-to-directory>
```

如下面的命令，将/data/configs/nginx/conf.d/目录下的所有文件都保存于**nginx-config-files**对象中：

```
~]$ kubectl create configmap nginx-config-files --from-file=./data/configs/nginx/conf.d/
```

此目录中包含**myserver.conf**、**myserver-status.cfg**和**myserver-gzip.cfg**三个配置文件。创建**ConfigMap**资源时，它们会被分别存储为三个键值数据，如下面的命令及其结果所示：

```
~]$ kubectl get cm nginx-config-files -o yaml
apiVersion: v1
data:
  myserver-gzip.cfg: |
    gzip on;
    gzip_comp_level 5;
    gzip_proxied      expired no-cache no-store private auth;
    gzip_types text/plain text/css application/xml text/javascript;
  myserver-status.cfg: |
    location /nginx-status {
      stub_status on;
      access_log off;
    }
  myserver.conf: |
    server {
      listen 8080;
      server_name www.ilinux.io;

      include /etc/nginx/conf.d/myserver-*.cfg;

      location / {
        root /usr/share/nginx/html;
      }
    }
kind: ConfigMap
.....
```

注意，**describe**命令和**get-o yaml**命令都可显示由文件创建而成的键及其值，不过两者使用的键和值之间的分隔符不同。

4.使用清单创建

基于配置文件创建**ConfigMap**资源时，它所使用的字段包括通常的**apiVersion**、**kind**和**metadata**字段，以及用于存储数据的关键字段“**data**”。例如下面的示例代码：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-demo
  namespace: default
data:
  log_level: INFO
  log_file: /var/log/test.log
```

如果其值来自于文件内容时，则使用配置文件创建**ConfigMap**资源的便捷性还不如直接通过命令行的方式，因此建议直接使用命令行加载文件或目录的方式进行创建。为了便于配置留存，可以在创建完成后使用**get-o yaml**命令获取到相关信息后再进行编辑留存。

8.4.2 向Pod环境变量传递ConfigMap对象键值数据

如8.3节中所描述的，Pod资源的环境变量值的获得方式之一包括引用ConfigMap对象中的数据，这一点通过在env字段中为valueFrom内嵌configMapKeyRef对象即可实现，其使用格式如下：

```
valueFrom:
  configMapKeyRef:
    key:
    name:
    optional:
```

其中，字段name的值为要引用的ConfigMap对象的名称，字段key可用于指定要引用ConfigMap对象中某键的键名，而字段optional则用于为当前Pod资源指明此引用是否为可选。此类环境变量的使用方式与直接定义的环境变量并无区别，它们可被用于容器的启动脚本或直接传递给容器应用等。

下面是保存于配置文件configmap-env.yaml的资源定义示例，它包含了两个资源，彼此之间使用“---”相分隔。第一个资源是名为busybox-httpd-config的ConfigMap对象，它包含了两个键值数据；第二个资源是名为configmap-env-demo的Pod对象，它通过环境变量引用了busybox-httpd-config对象中的键值数据，并将其直接传递给了自定义运行的容器应用httpd：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: busybox-httpd-config
  namespace: default
data:
  httpd_port: "8080"
  verbose_level: "-vv"
---
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env-demo
  namespace: default
spec:
```

```
containers:
- image: busybox
  name: busybox-httpd
  command: ["/bin/httpd"]
  args: ["-f", "-p", "$(HTTPD_PORT)", "$(HTTPD_LOG_VERBOSE)"]
  env:
  - name: HTTPD_PORT
    valueFrom:
      configMapKeyRef:
        name: busybox-httpd-config
        key: httpd_port
  - name: HTTPD_LOG_VERBOSE
    valueFrom:
      configMapKeyRef:
        name: busybox-httpd-config
        key: verbose_level
    optional: true
```

注意，在`command`或`args`字段中引用环境变量要使用“\$(`VAR_NAME`)”的格式。待上面配置文件中的资源创建完成后，可以通过如下命令验证Pod资源监听的端口等配置信息是否为`busybox-httpd-config`中定义的内容：

```
~]$ kubectl exec configmap-env-demo ps aux
PID   USER     TIME  COMMAND
1  root      0:00  /bin/httpd -f -p 8080 -vv
```



注意 创建引用了`ConfigMap`资源的Pod对象时，被引用的资源必须事先存在，否则将无法启动相应的容器，直到被依赖的资源创建完成为止。不过，那些未引用不存在的`ConfigMap`资源的容器将不受此影响。另外，`ConfigMap`是名称空间级别的资源，它必须与引用它的Pod资源在同一空间中。

假设存在这么一种情形，某`ConfigMap`资源中存在较多的键值数据，而全部或大部分的这些键值数据都需要由容器来引用。此时，为容器逐一配置相应环境变量将是一件颇为劳心费神之事，而且极易出错。对此，Pod资源支持在容器中使用`envFrom`字段直接将`ConfigMap`资源中的所有键值一次性地完成导入。它的使用格式如下：

```
spec:
  containers:
  - image: some-image
```

```
envFrom:
- prefix <string>
  configMapRef:
    name <string>
    optional <boolean>
```

envFrom字段值是对象列表，可用于同时从多个**ConfigMap**对象导入键值数据。为了避免从多个**ConfigMap**引用键值数据时产生键名冲突，可以在每个引用中将被导入的键使用**prefix**字段指定一个特定的前缀，如“HTCFG_”一类的字符串，于是，**ConfigMap**对象中的**httpd_port**将成为**Pod**资源中名为**HTCFG_httpd_port**的变量。



注意 如果键名中使用了连接线“-”，那么在转换为变量名时，连接线将被自动替换为下划线“_”。

例如，把上面示例中的**Pod**资源转为如下形式的定义（**configmap-envfrom-pod.yaml**配置文件）后，其引用**ConfigMap**进行配置的效果并无不同：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-envfrom-demo
  namespace: default
spec:
  containers:
  - image: busybox
    name: busybox-httpd
    command: ["/bin/httpd"]
    args: ["-f", "-p", "${HTCFG_httpd_port}", "${HTCFG_verbose_level}"]
    envFrom:
    - prefix: HTCFG_
      configMapRef:
        name: busybox-httpd-config
        optional: false
```

待**Pod**资源创建完成后，可通过查看其环境变量验证其导入的结果：

```
~]$ kubectl exec configmap-envfrom-demo printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=configmap-envfrom-demo
HTCFG_httpd_port=8080
HTCFG_verbose_level=-vv
```

值得提醒的是，从`ConfigMap`对象导入资源时，`prefix`为可选字段，省略时，所有变量名同`ConfigMap`中的键名。如果不存在键名冲突的可能性，例如从单个`ConfigMap`对象导入变量或在`ConfigMap`对象中定义键名时已然添加了特定的前缀，那么省略前缀的定义既不会导致键名冲突，又能保持变量的简洁。

8.4.3 ConfigMap存储卷

若ConfigMap对象中的键值来源于较长的文件内容，那么使用环境变量将其导入会使得变量值占据过多的内存空间而且不易处理。此类数据通常用于为容器应用提供配置文件，因此将其内容直接作为文件进行引用方为较好的选择。其实现方式是，在定义Pod资源时，将此类ConfigMap对象配置为ConfigMap类型的存储卷，而后由容器将其挂载至特定的挂载点后直接进行访问。

1. 挂载整个存储卷

关联为Pod资源的存储卷时，ConfigMap对象中的每个键都对应地表现为一个文件，键名转为文件名，而键值则为相应文件的内容，即便是通过直接值创建的键值数据，也一样表现为文件视图。挂载于容器上之后，由键值数据表现出的文件位于挂载点目录中，容器中的进程可直接读取这些文件的内容。

配置Pod资源时，基于存储卷的方式引用ConfigMap对象的方法非常简单，仅需要指明存储卷名称及要引用的ConfigMap对象名称即可。下面是于配置文件configmap-volume-pod.yaml中定义的Pod资源，它引用了8.4.1节下第3小节中创建的ConfigMap对象nginx-config-files，容器nginx-server将其挂载至应用程序Nginx加载配置文件模块的目录/etc/nginx/conf.d中，具体如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-demo
  namespace: default
spec:
  containers:
    - image: nginx:alpine
      name: nginx-server
      volumeMounts:
        - name: ngxconfig
          mountPath: /etc/nginx/conf.d/
          readOnly: true
  volumes:
    - name: ngxconfig
      configMap:
        name: nginx-config-files
```

此Pod资源引用的nginx-config-files中包含三个配置文件，其中myserver.conf定义了一个虚拟主机www.ilinux.io，并通过include指令包含/etc/nginx/conf.d/目录下以“myserver-”为前缀且以.cfg为后缀的所有配置文件，例如在nginx-config-files中包含的myserver-status.cfg和myserver-gzip.cfg。具体的配置内容请参考8.4.1节下第3小节中命令的输出。ConfigMap存储卷中的文件如图8-2所示。



图8-2 ConfigMap存储卷中的文件

创建此Pod资源后于Kubernetes集群中的某节点直接向Pod IP的8080端口发起访问请求，即可验证由nginx-config-files资源提供的配置信息是否生效，例如，通过/nginx-status访问其内建的stub status:

```
-]$ POD_IP=$(kubectl get pods configmap-volume-demo -o go-template={{.status.  
  podIP}})  
-]$ curl http://${POD_IP}:8080/nginx-status  
Active connections: 1  
server accepts handled requests  
  3 3 3  
Reading: 0 Writing: 1 Waiting: 0
```

当然，我们也可以直接于Pod资源的相应容器上执行命令来确认文件是否存在于挂载点目录中:

```
-]$ kubectl exec configmap-volume-demo ls /etc/nginx/conf.d/  
myserver-gzip.cfg  
myserver-status.cfg  
myserver.conf
```

进一步地，还可以于容器中运行Nginx的配置测试及打印命令，确认由ConfigMap资源提供的配置信息已然生效：

```
~]$ kubectl exec configmap-volume-demo -- nginx -T
.....
# configuration file /etc/nginx/conf.d/myserver.conf:
server {
    listen 8080;
    server_name www.ilinux.io;

    include /etc/nginx/conf.d/myserver-*.cfg;

    location / {
        root /usr/share/nginx/html;
    }
}

# configuration file /etc/nginx/conf.d/myserver-gzip.cfg:
gzip on;
gzip_comp_level 5;
gzip_proxied expired no-cache no-store private auth;
gzip_types text/plain text/css application/xml text/javascript;

# configuration file /etc/nginx/conf.d/myserver-status.cfg:
location /nginx-status {
    stub_status on;
    access_log off;
}
```

由上面两个命令的结果可见，**nginx-config-files**中的三个文件都被添加到了容器中，并且实现了由容器应用Nginx加载并生效。

2.挂载存储卷中的部分键值

有时候，用户很可能不期望在容器中挂载某ConfigMap存储卷后于挂载点目录导出所有的文件，这在通过一个ConfigMap对象为单个Pod资源中的多个容器分别提供配置时尤其常见。例如前面的示例中，用户可能只期望在容器中挂载ConfigMap存储卷后只“导出”其中的**myserver.conf**和**myserver-gzip.cfg**，只提供页面传输压缩功能，而不输出**nginx stub status**信息，此时将其**volumes**配置段改为如下所示的内容即可。为了以示区别并在后文中便于引用及说明问题，这里将其保存于单独的配置文件**configmap-volume-demo-2.yaml**中，并将Pod资源命名为**configmap-volume-demo-2**：

```
apiVersion: v1
kind: Pod
metadata:
```

```
name: configmap-volume-demo-2
namespace: default
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: ngxconfig
      mountPath: /etc/nginx/conf.d/
      readOnly: true
  volumes:
  - name: ngxconfig
    configMap:
      name: nginx-config-files
      items:
      - key: myserver.conf
        path: myserver.conf
        mode: 0644
      - key: myserver-gzip.cfg
        path: myserver-compression.cfg
```

`configMap`存储卷的`items`字段的值是一个对象列表，可嵌套使用的字段有三个，具体如下。

·`key<string>`：要引用的键名称，必选字段。

·`path<string>`：对应的键于挂载点目录中生成的文件的相对路径，可以不同于键名称，必选字段。

·`mode<integer>`：文件的权限模型，可用范围为0到0777。

上面的配置示例中，`myserver-gzip.cfg`映射成了`myserver-compression.cfg`文件，而`myserver.conf`则保持了与键名同名，并明确指定使用0644的权限，从而达成了仅装载部分文件至容器之目的。

3.独立挂载存储卷中的键值

前述方式中，无论是装载所有文件还是部分文件，挂载点目录下原有的文件都会被隐藏。对于期望将`ConfigMap`对象提供的配置文件补充于挂载点目录下的需求来说，这种方式显然难以如愿。例如，`/etc/nginx/conf.d`目录中原本就存在一些文件（如`default.conf`），用户期望将`nginx-config-files`中的全部或部分文件装载进此目录中而不影响其原有的文件。

事实上，此种需求可以通过此前第7章中曾经于容器的`volumeMounts`字段中使用的`subPath`字段来解决，它可以支持用户从存

储卷挂载单个文件或单个目录而非整个存储卷。例如，下面的示例就于/etc/nginx/conf.d目录中单独挂载了两个文件，而保留于了目录下原有的文件。为了以示区别并在后文中便于引用及说明问题，这里将其保存于单独的配置文件configmap-volume-demo-3.yaml中，并将Pod资源命名为configmap-volume-demo-3:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-demo-3
  namespace: default
spec:
  containers:
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: ngxconfig
          mountPath: /etc/nginx/conf.d/myserver.conf
          subPath: myserver.conf
          readOnly: true
        - name: ngxconfig
          mountPath: /etc/nginx/conf.d/myserver-status.cfg
          subPath: myserver-status.cfg
          readOnly: true
  volumes:
    - name: ngxconfig
      configMap:
        name: nginx-config-files
```

基于上述配置创建了Pod资源之后，即可通过命令验证/etc/nginx/conf.d目录中的原有文件确实能够得以保留，如下面的命令及其结果所示:

```
~]$ kubectl exec configmap-volume-demo-3 ls /etc/nginx/conf.d
default.conf
myserver-gzip.cfg
myserver.conf
```

8.4.4 容器应用重载新配置

相较于环境变量来说，使用ConfigMap资源为容器应用提供配置的优势之一在于其支持容器应用动态更新其配置：用户直接更新ConfigMap对象，而后由容器应用重载其配置文件即可。

细心的读者或许已经发现，挂载ConfigMap存储卷的挂载点目录中的文件都是符号链接，它们指向了当前目录中的“..data”，而“..data”也是符号链接，它指向了名字形如“..2018_03_17_14_41_41.256460087”的目录，这个目录才是存储卷的真正挂载点。例如，查看8.4.3节下第1小节中创建的Pod资源容器中的挂载点中的文件列表，它将显示出类似如下结果：

```
~]$ kubectl exec -it configmap-volume-demo -- ls -lA /etc/nginx/conf.d
total 0
drwxr-xr-x  2 root   root   79 Mar 17 14:41 ..2018_03_17_14_41_41.256460087
lrwxrwxrwx  1 root   root   31 Mar 17 14:41 ..data ->
..2018_03_17_14_41_41.256460087
lrwxrwxrwx  1 root   root   24 Mar 17 10:19 myserver-gzip.cfg -> ..data/myserver-
gzip.cfg
lrwxrwxrwx  1 root   root   26 Mar 17 10:19 myserver-status.cfg ->
..data/myserver-
status.cfg
lrwxrwxrwx  1 root   root   20 Mar 17 10:19 myserver.conf -> ..data/myserver.conf
```

这样两级符号链接设定的好处在于，在引用的ConfigMap对象中的数据发生改变时，它将被重新挂载至一个新的临时目录下，而后“..data”将指向此新的挂载点，便达到了同时更新存储卷上所有文件数据之目的。例如，使用kubectl edit命令直接在ConfigMap对象nginx-config-files中的myserver-status.cfg配置段中增加“allow 127.0.0.0/8;”和“deny all;”两行，而后再次查看configmap-volume-demo中的容器的挂载点目录中的文件列表，结果是其挂载点已经指向了新的位置，例如下面的命令及其结果所示：

```
~]$ kubectl exec -it configmap-volume-demo -- ls -lA /etc/nginx/conf.d
total 0
rwxr-xr-x   2 root   root   79 Mar 17 15:17 ..2018_03_17_15_17_11.170451948
lrwxrwxrwx  1 root   root   31 Mar 17 15:17 ..data ->
..2018_03_17_15_17_11.170451948
lrwxrwxrwx  1 root   root   24 Mar 17 10:19 myserver-gzip.cfg -> ..data/myserver-
gzip.cfg
```

```
lrwxrwxrwx 1 root root 26 Mar 17 10:19 myserver-status.cfg ->
..data/myserver-
status.cfg
lrwxrwxrwx 1 root root 20 Mar 17 10:19 myserver.conf -> ..data/myserver.conf
```

此时，若要使容器中应用程序的新配置生效，则需要于Pod资源的相应容器上执行配置重载操作。例如，Nginx可通过其“nginx-s reload”命令完成配置文件重载，如下面的命令所示：

```
~]$ kubectl exec configmap-volume-demo -- nginx -s reload
2018/08/17 15:18:19 [notice] 222#222: signal process started
```

此时，如果于此容器之外的位置访问/nginx-status页面的请求是被拒绝的，则表明新配置已然生效，如下面的命令及其结果所示：

```
~]$ curl http://${POD_IP}:8080/nginx-status
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.13.9</center>
</body>
</html>
```

然而，需要注意的是，对于不支持配置文件重载操作的容器应用来说，只有那些在ConfigMap对象更新后创建的Pod资源中的容器会应用到新配置，此时如果不重启旧有的容器，则会导致配置不一致的问题。即使对于支持重载操作的应用来说，由于新的配置信息并非同步推送进所有容器中，而且各容器的重载操作也未必能同时进行，因此在更新时，短时间内仍然会存在配置不一致的现象。

另外，使用8.4.3下第3小节中的方式独立挂载存储卷中的文件的容器，其挂载配置文件的方式并非是以两级链接的方式进行的，因此存储卷无法确保所有挂载的文件可以被同时更新至容器中，因此为了确保配置信息的一致性，目前这种类型的挂载不支持文件更新操作。读者可对此自行进行验证。

8.4.5 使用ConfigMap资源的注意事项

在Pod资源中调用ConfigMap对象时需要注意以下几个问题。

- 以存储卷方式引用的ConfigMap必须先于Pod存在，除非在Pod中将它们全部标记为“optional”，否则将会导致Pod无法正常启动的错误；同样，即使存在ConfigMap，在引用的键不存在时，也会导致一样的错误。

- 当以环境变量方式注入的ConfigMap中的键不存在时会被忽略，Pod可以正常启动，但错误引用的信息会以“InvalidVariableNames”事件记录于日志中。

- ConfigMap是名称空间级的资源，因此，引用它的Pod必须处于同一名称空间中。

- kubelet不支持引用Kubernetes API Server上不存在的ConfigMap，这包括那些通过kubelet的“--manifest-url”或“--config”选项，以及kubelet REST API创建的Pod。

8.5 Secret资源

Secret资源的功能类似于ConfigMap，但它专用于存放敏感数据，例如密码、数字证书、私钥、令牌和SSH key等。

8.5.1 Secret概述

Secret对象存储数据的方式及使用方式类似于ConfigMap对象，以键值方式存储数据，在Pod资源中通过环境变量或存储卷进行数据访问。不同的是，Secret对象仅会被分发至调用了此对象的Pod资源所在的工作节点，且只能由节点将其存储于内存中。另外，Secret对象的数据的存储及打印格式为Base64编码的字符串，因此用户在创建Secret对象时也要提供此种编码格式的数据。不过，在容器中以环境变量或存储卷的方式访问时，它们会被自动解码为明文格式。

需要注意的是，在Master节点上，Secret对象以非加密的格式存储于etcd中，因此管理员必须加以精心管控以确保敏感数据的机密性，必须确保etcd集群节点间以及与API Server的安全通信，etcd服务的访问授权，还包括用户访问API Server时的授权，因为拥有创建Pod资源的用户都可以使用Secret资源并能够通过Pod中的容器访问其数据。

Secret对象主要有两种用途，一是作为存储卷注入到Pod上由容器应用程序所使用，二是用于kubelet为Pod里的容器拉取镜像时向私有仓库提供认证信息。不过，后面使用ServiceAccount资源自建的Secret对象是一种更具安全性的方式。通过ConfigMap和Secret配置容器的方式如图8-3所示。

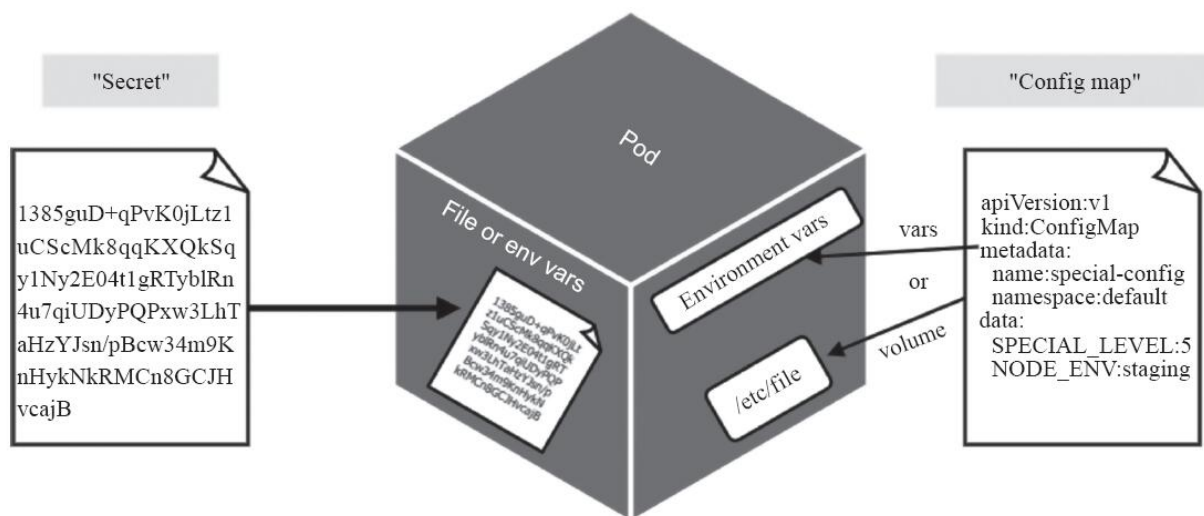


图8-3 通过ConfigMap和Secret配置容器

Secret资源主要由四种类型组成，具体如下。

- Opaque**: 自定义数据内容；base64编码，用来存储密码、密钥、信息、证书等数据，类型标识符为**generic**。

- kubernetes.io/service-account-token**: Service Account的认证信息，可在创建Service Account时由Kubernetes自动创建。

- kubernetes.io/dockerconfigjson**: 用来存储Docker镜像仓库的认证信息，类型标识为**docker-registry**。

- kubernetes.io/tls**: 用于为SSL通信模式存储证书和私钥文件，命令式创建时类型标识为**tls**。



注意 base64编码并非加密机制，其编码的数据可使用“base64--decode”一类的命令进行解码。

8.5.2 创建Secret资源

手动创建Secret对象的方式有两种：通过`kubectl create`命令和使用Secret配置文件。

1. 命令式创建

不少场景中，Pod中的应用需要通过用户名和密码访问其他服务，例如访问数据库系统等。创建此类的Secret对象，可以使用“`kubectl create secret generic<SECRET_NAME>--from-literal=key=value`”命令直接进行创建，不过为用户认证之需进行创建时，其使用的键名通常是username和password。例如下面的命令，以“root/ikubernetes”分别为用户名和密码创建了一个名为mysql-auth的Secret对象：

```
~]$ kubectl create secret generic mysql-auth --from-literal=username=root \
--from-literal=password=ikubernetes
```

而后即可查看新建资源的属性信息，由下面的命令及其输出结果可以看出，以generic标识符创建的Secret对象是为Opaque类型，其键值数据会以Base64的编码格式进行保存和打印：

```
~]$ kubectl get secrets mysql-auth -o yaml
apiVersion: v1
data:
  password: aWt1YmVybmV0ZXM=
  username: cm9vdA==
kind: Secret
metadata:
  .....
type: Opaque
```

不过，Kubernetes系统的Secret对象的Base64编码的数据并非加密格式，许多相关的工具程序均可轻松完成解码，如下面所示的Base64命令：

```
~]$ echo aWt1YmVybmV0ZXM= | base64 -d
ikubernetes
```

对于本身业已存储于文件中的数据，也可以在创建generic格式Secret对象时使用“--from-file”选项从文件中直接进行加载，例如创建用于SSH认证的Secret对象时，如果尚且没有认证信息文件，则需要首先使用命令生成一对认证文件：

```
~]$ ssh-keygen -t rsa -P '' -f ${HOME}/.ssh/id_rsa
```

而后使用“`kubectl create secret generic<SECRET_NAME>--from-file[=KEY1]=/PATH/TO/FILE`”命令加载文件内容并生成为Secret对象：

```
~]$ kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=${HOME}/.ssh/id_rsa --from-file=ssh-publickey=${HOME}/.ssh/id_rsa.pub
```

另外，若要基于私钥和数字证书文件创建用于SSL/TLS通信的Secret对象，则需要使用“`kubectl create secret tls<SECRET_NAME>--cert==key=`”命令来进行，注意其类型标识符为TLS。例如，假设需要为Nginx测试创建SSL虚拟主机，用户首先使用了类似如下的命令生成了私钥和自签证书：

```
~]$ (umask 077; openssl genrsa -out nginx.key 2048)
~]$ openssl req -new -x509 -key nginx.key -out nginx.crt \
-subj /C=CN/ST=Beijing/L=Beijing/O=DevOps/CN=www.ilinux.io
```

而后即可使用如下命令将这两个文件创建为Secret对象。需要注意的是，无论用户提供的证书和私钥文件使用的是什么名称，它们一律会被转换为分别以`tls.key`（私钥）和`tls.crt`（证书）为其键名：

```
~]$ kubectl create secret tls nginx-ssl --key=./nginx.key --cert=./nginx.crt
secret "nginx-ssl" created
```

注意其类型应该为“`kubernetes.io/tls`”，例如下面命令结果中的显示：

```
~]$ kubectl get secrets nginx-ssl
NAME          TYPE          DATA  AGE
nginx-ssl     kubernetes.io/tls  2      37s
```

由上述操作过程可见，命令式创建Secret对象与ConfigMap对象的方式几乎没有明显区别。

2. 清单式创建

Secret资源是标准的Kubernetes API对象，除了标准的apiVersion、kind和metadata字段，它可用的其他字段具体如下。

- data<map[string]string>: “key: value”格式的数据，通常是敏感信息，数据格式需是以Base64格式编码的字符串，因此需要用户事先完成编码。

- stringData<map[string]string>: 以明文格式（非Base64编码）定义的“key: value”数据；无须用户事先对数据进行Base64编码，而是在创建为Secret对象时自动进行编码并保存于data字段中；stringData字段中的明文不会被API Server输出，不过若是使用“kubectl apply”命令进行的创建，那么注解信息中还是可能会直接输出这些信息的。

- type<string>: 仅是为了便于编程方式处理Secret数据而提供的类型标识。

下面是保存于配置文件secret-demo.yaml中的Secret资源定义示例，其使用stringData提供了明文格式的键值数据，从而免去了事先进行手动编码的麻烦：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-demo
stringData:
  username: redis
  password: redis@ss
type: Opaque
```

Secret对象也是Kubernetes系统的“一等公民”，因此，使用标准资源创建命令即可完成其创建。相比较来说，基于清单文件将保存于文

件中的敏感信息创建Secret对象时，用户首先需要将敏感信息读出，转为Base64编码格式，而后再将其创建为清单文件，过程烦琐，反不如命令式创建来得便捷。不过，如果存在多次创建或重构之需，那么将其保存为配置清单也是情势所需。

8.5.3 Secret存储卷

类似于Pod消费ConfigMap对象的方式，Secret对象可以注入为环境变量，也可以存储为卷形式挂载使用。不过，容器应用通常会在发生错误时将所有环境变量保存于日志信息中，甚至有些应用在启动时即会将运行环境打印到日志中；另外，容器应用调用第三方程序为子进程时，这些子进程能够继承并使用父进程的所有环境变量。有鉴于此，使用环境变量引用Secret对象中的敏感信息实在算不上明智之举。

在Pod中使用Secret存储卷的方式，除了其类型及引用标识要替换为Secret及secretName之外，几乎完全类似于ConfigMap存储卷，包括支持使用挂载整个存储卷、只挂载存储卷中的指定键值以及独立挂载存储卷中的键等使用方式。

下面是定义在配置文件secret-volume-pod.yaml中的Secret资源使用示例，它将nginx-ssl关联为Pod资源的名为nginxcert的Secret存储卷，而后由容器web-server挂载至/etc/nginx/ssl目录下：

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-demo
  namespace: default
spec:
  containers:
    - image: nginx:alpine
      name: web-server
      volumeMounts:
        - name: nginxcert
          mountPath: /etc/nginx/ssl/
          readOnly: true
  volumes:
    - name: nginxcert
      secret:
        secretName: nginx-ssl
```

将上面资源清单文件中定义的资源创建于Kubernetes系统上，而后再查看容器挂载点目录中的文件，以确认其挂载是否成功完成。下面命令的结果显示，私钥文件tls.key和证书文件tls.crt已经成功保存于挂载点路径之下：

```
~]$ kubectl exec secret-volume-demo ls /etc/nginx/ssl/  
tls.crt  
tls.key
```

此时，通过ConfigMap对象为容器应用Nginx提供HTTPS虚拟主机配置，它只要使用由Secret对象生成的私钥和证书文件，即可定义出容器化运行的Nginx服务。

8.5.4 imagePullSecret资源对象

imagePullSecret资源可用于辅助kubelet从需要认证的私有镜像仓库获取镜像，它通过将Secret提供的密码传递给kubelet从而在拉取镜像前完成必要的认证过程。

使用imagePullSecret的方式有两种：一是创建docker-registry类型的Secret对象，并在定义Pod资源时明确通过“imagePullSecrets”字段给出；另一个是创建docker-registry类型的Secret对象，将其添加到某特定的ServiceAccount对象中，那些使用该ServiceAccount资源创建的Pod对象，以及默认使用该ServiceAccount的Pod对象都将会直接使用imagePullSecrets中的认证信息。由于尚未介绍到ServiceAccount资源，因此这里先介绍第一种方式的用法。

创建docker-registry类型的Secret对象时，要使用“`kubectl create secret docker-registry<SECRET_NAME>--docker-user=<USERNAME>--docker-password=<PASSWORD>--docker-email=<DOCKER_USER_EMAIL>`”的命令格式，其中的用户名、密码及邮件信息是在使用docker login命令登录时使用的认证信息。例如，下面的命令创建了名为local-registry的image pull secret对象：

```
~]$ kubectl create secret docker-registry local-registry --docker-username=Ops \
--docker-password=0pspass --docker-email=ops@ilinux.io
```

此类Secret对象打印的类型信息为“`kubernetes.io/dockerconfigjson`”，如下面的命令结果所示：

```
~]$ kubectl get secrets local-registry
```

NAME	TYPE	DATA	AGE
local-registry	kubernetes.io/dockerconfigjson	1	7s

而后，使用相应的私有registry中镜像的Pod资源的定义，即可通过imagePullSecrets字段使用此Secret对象，使用示例如下面的配置清单所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-imagepull-demo
  namespace: default
spec:
  imagePullSecrets:
    - name: local-registry
  containers:
    - image: registry.iilinux.io/dev/myimage
      name: myapp
```

上面的配置清单仅是一个示例，付诸运行时，需要由读者将其**Secret**中的内容及清单资源的镜像等信息的定义修改为实际可用的信息。

试想，正在运行的多数容器的镜像均来自于私有仓库时，为每个**Pod**资源显式定义**imagePullSecrets**实在不是一个好主意。好在还有基于**ServiceAccount**的**image pull secret**可用。

8.6 本章小结

本章的核心目标在于为读者说明配置容器应用的常用方式，这将在将同一**Pod**资源分别以不同的配置运行于不同的环境中时特别有用。本章主要描述了如下几种配置容器化应用的方式。

- 自定义命令行选项，为容器化应用传递特定参数。
- 通过环境变量向容器注入自定义数据。
- 基于**ConfigMap**对象，以环境变量或存储卷的形式向容器提供配置信息。
- 借助**Secret**对象，以存储卷的形式向容器应用提供敏感信息。

第9章 StatefulSet控制器

应用程序存在“有状态”和“无状态”两种类别，因为无状态类应用的Pod资源可按需增加、减少或重构，而不会对由其提供的服务产生除了并发响应能力之外的其他严重影响。Pod资源的常用控制器中，Deployment、ReplicaSet和DaemonSet等常用于管理无状态应用。但实际情况是，应用本身就是分布式的集群，各应用实例彼此之间存在着关联关系，甚至是次序、角色方面的相关性，其中的每个实例都有其自身的独特性而无法轻易由其他实例所取代。本章的主要目标就是描述专用于管理此类应用的Pod资源的控制器StatefulSet。

9.1 StatefulSet概述

ReplicaSet控制器可用于管控无状态应用，例如提供静态内容服务的Web服务器程序等，而对于有状态应用的管控，则是另一项专用控制器的任务—StatefulSet。

9.1.1 Stateful应用和Stateless应用

应用程序与用户、设备、其他应用程序或外部组件进行通信时，根据其是否需要记录前一次或多次通信中的相关事件信息以作为下一次通信的分类标准，可以将那些需要记录信息的应用程序称为有状态（**stateful**）应用，而无须记录的则称为无状态（**stateless**）应用。下面我们先来了解下状态和存储的关系。

- 状态是进程的时间属性。无状态意味着一个进程不必跟踪过去的交互操作，本质上可以说它是一个纯粹的功能性行为。对应地，有状态则意味着进程存储了以前交互过程的记录，并且可以基于它对新的请求进行响应。至于状态信息被保存在内存中，或者持久保存于磁盘上，则是另外一个问题。

- 存储是表述持久保存数据的方法，现今通常是指机械硬盘或SSD设备。若进程仅需操作内存中的数据，则表示其无须进行磁盘I/O操作；如果产生了I/O操作，则通常意味着数据的只读访问或读写访问行为。

如图9-1所示，将状态和存储这两个概念正交于坐标系中，则可以归结出如下几种应用程序类型。

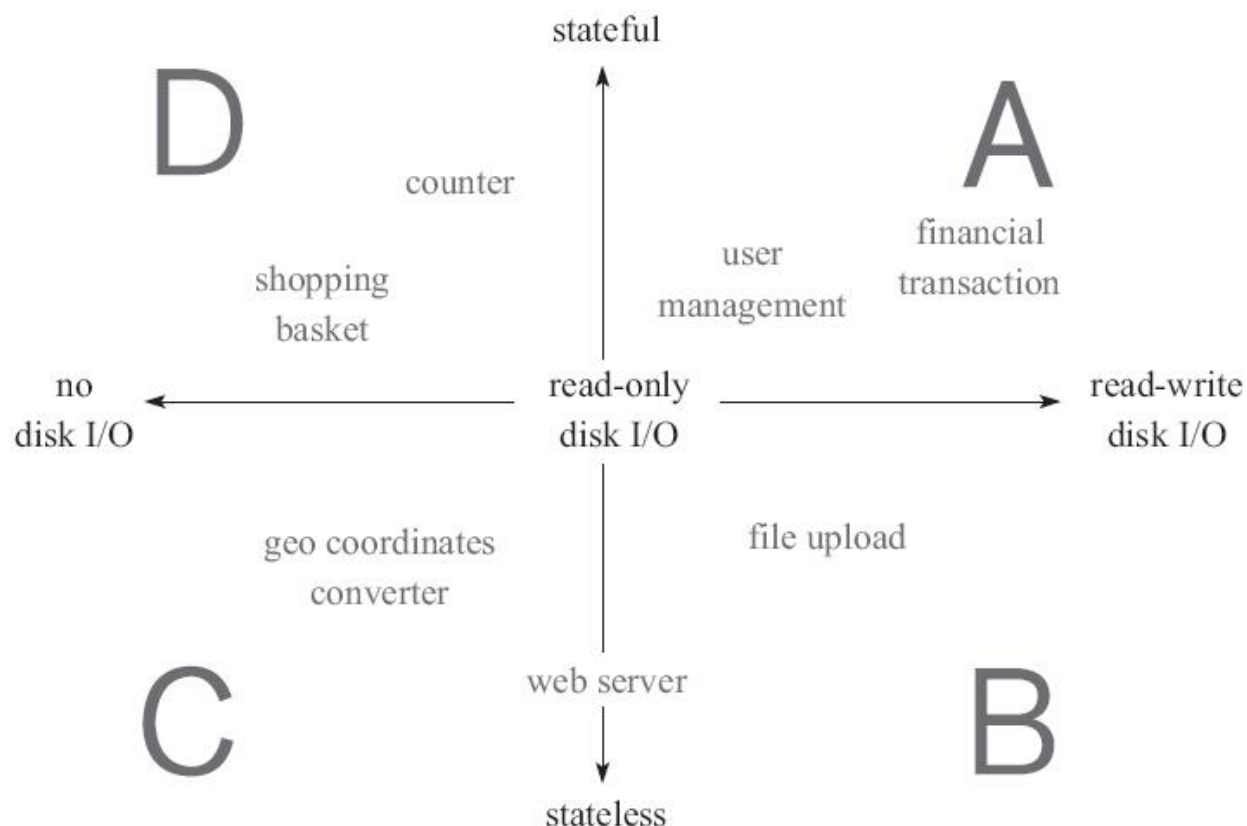


图9-1 状态和存储的关系

·象限A中是那些具有读写磁盘需求的有状态应用程序，如支持事务功能的各种RDBMS存储系统；另外各种分布式存储系统也是此类应用程序的典型，如Redis Cluster、MongoDB、ZooKeeper和Cassandra等。

·象限B中包含两类应用程序：一类是那些具有读写磁盘需求的无状态应用程序，如具有幂等性的文件上传类服务程序；另一类是仅需只读类I/O访问的无状态应用程序，例如，从外部存储加载静态资源以响应用户请求的Web服务程序。

·象限C中是无磁盘访问需求的无状态应用程序，如地理坐标转换器应用。

·象限D中是无磁盘访问需求的有状态应用程序，如电子商城程序中的购物车系统。

不过，用户拥有放置应用程序的部分自由度，例如，使用购物车的电子商城系统中，一般需要确保购物车里的物品在整个会话期间均

保持可用状态，因此它可能不允许使用纯内存的解决方案。另外，设计有状态应用程序时需要着重考虑的另一个方面是数据持久存储的位置，在应用程序所在的节点发生故障后依然需要确保数据可被访问的场景就需要一个外部的持久存储系统，否则使用节点本地存储卷即可。

9.1.2 StatefulSet控制器概述

前面章节中曾讲到，**ReplicaSet**控制器能够从一个预置的**Pod**模板创建一个或多个**Pod**资源，除了主机名和IP地址等属性之外，这些**Pod**资源并没有本质上的区别，就连它们的名称也是使用同一种散列模式生成，具有很强的相似性。通常，每一个访问请求都会以与其他请求相隔离的方式被这类应用所处理，不分先后也无须关心它们是否存在关联关系，哪怕它们先后来自于同一个请求者。于是，任何一个**Pod**资源都可以被**ReplicaSet**控制器重构出的新版本所替代，管理员更多关注的也是它们的群体特征，而无须过于关注任何一个个体。提供静态内容服务的**Web**服务器程序是这类应用的典型代表之一。

对应地，另一类应用程序在处理客户端请求时，对当前请求的处理需要以前一次或多次的请求为基础进行，新客户端发起的请求则会被其施加专用标识，以确保其后续的请求可以被识别。电商或社交等一类**Web**应用站点中的服务程序通常属于此类应用。另外还包含了以更强关联关系处理请求的应用，例如，**RDBMS**系统上处于同一个事务中的多个请求不但彼此之间存在关联性，而且还要以严格的顺序执行。这类应用一般需要记录请求连接的相关信息，即“状态”，有的甚至还需要持久保存由请求生成的数据，尤其是存储服务类的应用，运行于**Kubernetes**系统上时需要用到持久存储卷。

若**ReplicaSet**控制器在**Pod**模板中包含了某**PVC**（**Persistent Volume Claim**）的引用，则由它创建的所有**Pod**资源都将共享此存储卷。**PVC**后端的**PV**访问模型配置为**Read-OnlyMany**或**ReadWriteMany**时，这些**Pod**资源中的容器应用挂载存储卷后也就有了相同的数据集。不过，大多数情况是，一个集群系统的分布式应用中，每个实例都有可能需要存储使用不同的数据集，或者各自拥有其专有的数据副本，例如，分布式文件系统**GlusterFS**和分布式文档存储**MongoDB**中的每个实例各自使用专有的数据集，分布式服务框架**ZooKeeper**以及主从复制集群中的**Redis**的每个实例各自拥有其专用的数据副本。由于**ReplicaSet**控制器使用同一个模板生成**Pod**资源，显然，它无法实现为每个**Pod**资源创建专用的存储卷。别的可考虑使用的方案中，自主式**Pod**资源没有自愈能力，而组织多个只负责生成一个**Pod**资源的**ReplicaSet**控制器则有规模扩展不便的尴尬。

进一步来说，除了要用到专有的持久存储卷之外，有些集群类的分布式应用实例在运行期间还存在角色上的差异，它们存在单向/双向的基于IP地址或主机名的引用关系，例如主从复制集群中的MySQL从节点对主节点的引用。这类应用实例，每一个都应当作为一个独立的个体对待。ReplicaSet对象控制下的Pod资源重构后，其名称和IP地址都存在变动的可能性，因此也无法适配此种场景之需。而StatefulSet（有状态副本集）则是专门用来满足此类应用的控制器类型，由其管控的每个Pod对象都有着固定的主机名和专有存储卷，即便被重构后亦能保持不变。

对比可见，ReplicaSet管控下的Pod资源更像是一群“家畜”（cattle），它们无状态，每个个体均被无区别地对待，因此也就可在任意时刻被另一个具有不同标识的同类事物所取代。而StatefulSet控制器治下的Pod资源更像是多个“宠物”（pet），每一个实例都有着其特有的状态，即使被重构，也得与其前任拥有相同的标识。事实上，在云原生应用的体系里有两组常用的近义词，第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）和可丢弃（disposable），它们都可用于表述无状态应用。另一组是有状态（stateful）、宠物（pet）、具名（having name）和不可丢弃（non-disposable），它们则都可用于称呼有状态应用。

自Kubernetes 1.3起开始通过PetSet控制器支持有状态应用，并于1.5版本中将其重命名为StatefulSet，支持每个Pod对象一个专有索引、有序部署、有序终止、固定的标识符及固定的存储卷等特性。不过在1.9版本之前，它一直处于beta级别。

9.1.3 StatefulSet的特性

StatefulSet是Pod资源控制器的一种实现，用于部署和扩展有状态应用的Pod资源，确保它们的运行顺序及每个Pod资源的唯一性。其与**ReplicaSet**控制器不同的是，虽然所有的Pod对象都基于同一个spec配置所创建，但**StatefulSet**需要为每个Pod维持一个唯一且固定的标识符，必要时还要为其创建专有的存储卷。**StatefulSet**主要适用于那些依赖于下列类型资源的应用程序。

- 稳定且唯一的网络标识符。
- 稳定且持久的存储。
- 有序、优雅地部署和扩展。
- 有序、优雅地删除和终止。
- 有序而自动地滚动更新。

一般来说，一个典型、完整可用的**StatefulSet**通常由三个组件构成：**Headless Service**、**StatefulSet**和**volumeClaimTemplate**。其中，**Headless Service**用于为Pod资源标识符生成可解析的DNS资源记录，**StatefulSet**用于管控Pod资源，**volumeClaimTemplate**则基于静态或动态的PV供给方式为Pod资源提供专有且固定的存储。

对于一个拥有N个副本的**StatefulSet**来说，其Pod对象会被有序创建，顺序依次是{0...N-1}，删除则以相反的顺序进行。不过，**Kubernetes 1.7**及其之后的版本也支持并行管理Pod对象的策略。Pod资源的名称格式为\$(statefulset name)-\$(ordinal)，例如，名称为Web的**ReplicaSet**资源所生成的Pod对象的名称依次为web-0、web-1、web-2等，其域名后缀可由相关的**Headless**类型的**Service**资源给出，格式为\$(service name).\$(namespace).svc.cluster.local，cluster.local是集群默认使用的域名。

ReplicaSet控制器会为每个**VolumeClaim**模板创建一个专用的PV，它会从模板中指定的**StorageClass**中为每个PVC创建PV，未指定时将使用默认的**StorageClass**资源，而如果存储系统不支持PV的动态供给，就

需要管理员事先创建好满足需求的所有PV。删除Pod资源甚至是ReplicaSet控制器并不会删除与其相关的PV资源以确保数据安全，它需要由用户手动删除。这也意味着，Pod资源被重新调度至其他节点时，其PV及数据可复用。Pod名称、PVC和PV关系如图9-2所示。

ReplicaSet也支持规模扩缩容操作，扩容意味着按索引顺序增加更多的Pod资源，而缩容则表示按逆序依次删除索引号最大的Pod资源直到规模数量满足目标设定值。执行扩容操作时，应用至某Pod对象之前必须确保其前的每个Pod对象都已经就绪，反之，终止一个Pod对象时，必须事先确保它的后继者已经终止完成。考虑到不少的有状态应用不支持规模的安全快速缩减，因此，ReplicaSet控制器不支持缩容时的并行操作，一次仅能终止一个Pod资源，以免导致数据讹误。这通常也意味着，存在错误的未恢复的Pod资源时，ReplicaSet控制器也会拒绝启动缩容操作。此外，缩容操作导致的Pod资源终止也不会删除与其相关的PV，以确保数据安全。

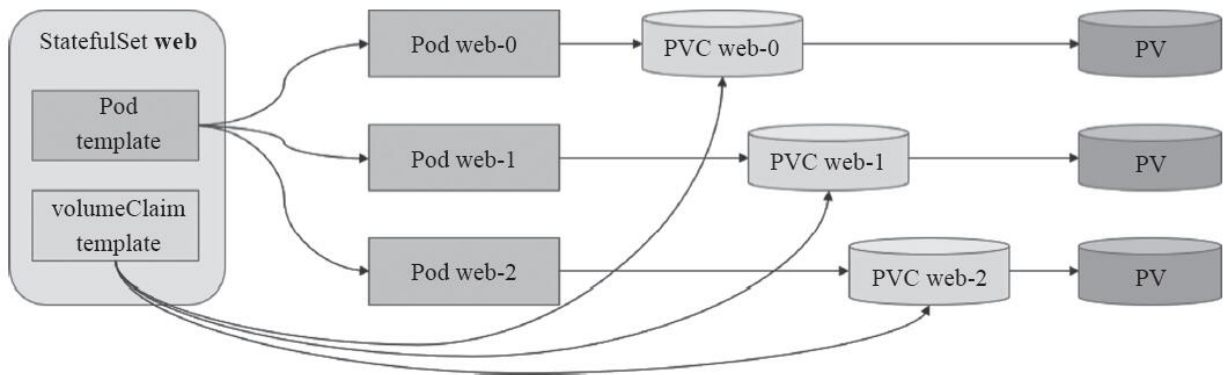


图9-2 Pod名称、PVC和PV

Kubernetes自1.7版本起还支持用户自定义更新策略，该版本兼容支持之前版本中的删除后更新（OnDelete）策略，以及新的滚动更新策略（RollingUpdate）。OnDelete意味着ReplicaSet不会自动更新Pod资源除非它被删除而激活重建操作。RollingUpdate是默认的更新策略，它支持Pod资源的自动、滚动更新。更新顺序与终止Pod资源的顺序相同，由索引号最大的资源开始，终止一个并完成其更新，而后更新下一个。另外，RollingUpdate还支持分区（partition）机制，用户可基于某个用于分区的索引号对Pod资源进行分区，所以大于等于此索引号的Pod资源会被滚动更新，如图9-3所示。而小于此索引号的Pod资源则不会被更新，即便是此范围内的某Pod资源被删除，它也一样会被基于旧版本的Pod模板重建。

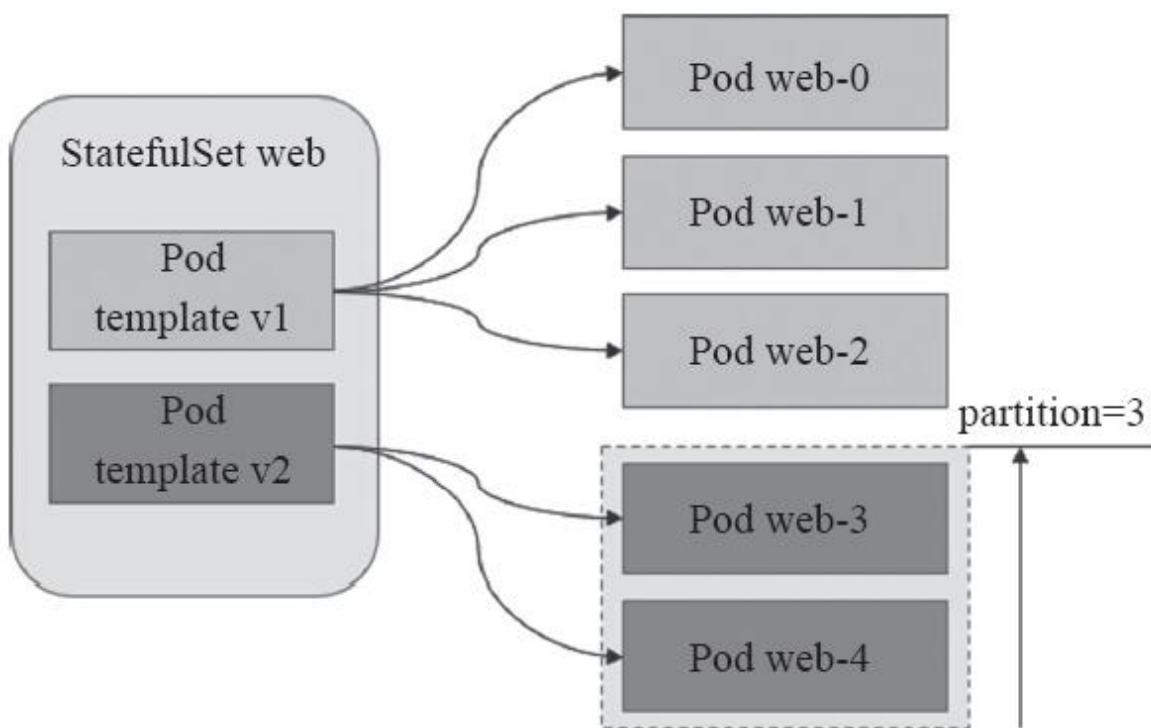


图9-3 ReplicaSet分区滚动更新

若给定的分区号大于副本数量，则意味着不会有Pod资源索引号大于此分区号，所有的Pod资源均不会被更新，对于暂存发布、金丝雀发布或分段发布来说，这也是有用的设定。

9.2 StatefulSet基础应用

本节将基于**StorageClass**及其相关的**PV**动态供给功能创建一个**stateful**资源示例，并验证它的各种特性。

9.2.1 创建StatefulSet对象

如前所述，一个完整的StatefulSet控制器需要由一个Headless Service、一个StatefulSet和一个volumeClaimTemplate组成。其中，Headless Service用于为Pod资源标识符生成可解析的DNS资源记录，StatefulSet用于管控Pod资源，volumeClaimTemplate则基于静态或动态的PV供给方式为Pod资源提供专有且固定的存储，如下面的资源清单中的定义所示：

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
  labels:
    app: myapp-svc
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: myapp-pod
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: myapp
spec:
  serviceName: myapp-svc
  replicas: 2
  selector:
    matchLabels:
      app: myapp-pod
  template:
    metadata:
      labels:
        app: myapp-pod
    spec:
      containers:
        - name: myapp
          image: ikubernetes/myapp:v5
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: myappdata
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: myappdata
      spec:
```



```
accessModes: [ "ReadWriteOnce" ]
storageClassName: "gluster-dynamic"
resources:
  requests:
    storage: 2Gi
```

上面示例中的配置定义在statefulset-demo.yaml文件中。由于StatefulSet资源依赖于一个事先存在的Headless类型的Service资源，因此，这里首先定义了一个名为myapp-svc的Headless Service资源，用于为关联到的每个Pod资源创建DNS资源记录。接着定义了一个名为myapp的StatefulSet资源，它通过Pod模板创建了两个Pod资源副本，并基于volumeClaimTemplates（存储卷申请模板）向gluster-dynamic存储类请求动态供给PV，从而为每个Pod资源提供大小为2GB的专用存储卷。

事实上，定义StatefulSet资源时，spec中必须要嵌套的字段为“serviceName”和“template”，用于指定关联的Headless Service和要使用的Pod模板，“volumeClaimTemplates”

字段用于为Pod资源创建专有存储卷PVC模板，它可内嵌使用的字段即为persistentVolumeClaim资源的可用字段，对StatefulSet资源为可选字段。

在依赖的存储类满足条件之后，即可创建清单中定义的相关资源：

```
~]$ kubectl apply -f statefulset-demo.yaml
service "myapp-svc" created
statefulset.apps "myapp" created
```

默认情况下，StatefulSet控制器以串行的方式创建各Pod副本，如果想要以并行方式创建和删除Pod资源，则可以设定.spec.podManagementPolicy字段的值为“Parallel”，默认值为“OrderedReady”。使用默认的顺序创建策略时，可以使用下面的命令观察相关Pod资源的顺次生成过程：

```
~]$ kubectl get pods -l app=myapp-pod -w
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-0	0/1	Pending	0	0s
myapp-0	0/1	ContainerCreating	0	0s

myapp-0	1/1	Running	0	42s
myapp-1	0/1	Pending	0	0s
myapp-1	0/1	ContainerCreating	0	0s
myapp-1	1/1	Running	0	14s

待所有的Pod资源都创建完成之后，可以在StatefulSet资源的相关状态中看到相关Pod资源的就绪信息：

```
~]$ kubectl get statefulsets myapp
```

NAME	DESIRED	CURRENT	AGE
myapp	2	2	10m

若上述资源的相关信息一切正常，则StatefulSet资源myapp已然就绪，其服务也可由其他依赖方所调用。

9.2.2 Pod资源标识符及存储卷

由StatefulSet控制器创建的Pod资源拥有固定、唯一的标识和专用存储卷，即便重新调度或终止后重建，其名称也依然保持不变，且此前的存储卷及其数据不会丢失。

1.Pod资源的固定标识符

如前所述，由StatefulSet控制器创建的Pod对象拥有固定且唯一的标识符，它们基于唯一的索引序号及相关的StatefulSet对象的名称而生成，格式为“<statefulset name>-<ordinal index>”，如下面的命令结果所示，两个Pod对象的名称分别为myapp-0和myapp-1，其中myapp为控制器的名称：

```
~]$ kubectl get pods -l app=myapp-pod
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-0	1/1	Running	0	4m
myapp-1	1/1	Running	0	3m

Pod资源的主机名同其资源名称，因此也是带索引序号的名称格式，如下面的命令结果所示：

```
~]$ for i in 0 1; do kubectl exec myapp-$i -- sh -c 'hostname'; done
```

myapp-0
myapp-1

这些名称标识会由StatefulSet资源相关的Headless Service资源创建为DNS资源记录，其域名格式为\$(service_name)\$. \$(namespace).svc.cluster.local，其中“cluster.local”是集群的默认域名。

在Pod资源创建后，与其相关的DNS资源记录格式为“\$(pod_name)\$. \$(service_name)\$. \$(namespace).svc.cluster.local”，

例如前面创建的两个Pod资源的资源记录为myapp-0.

myapp-svc.default.svc.cluster.local和myapp-1.myapp-svc.default.svc.cluster.local。下面再创建一个临时的Pod资源，并通过

其交互式接口测试上述两个名称的解析，其中粗体部分标识的内容为要运行的测试命令：

```
~]$ kubectl run -it --image busybox dns-client --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup myapp-0.myapp-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:       myapp-0.myapp-svc
Address 1: 10.244.1.25 myapp-0.myapp-svc.default.svc.cluster.local
/ # nslookup myapp-1.myapp-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:       myapp-1.myapp-svc
Address 1: 10.244.2.23 myapp-1.myapp-svc.default.svc.cluster.local
/ # nslookup myapp-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:       myapp-svc
Address 1: 10.244.1.25 myapp-0.myapp-svc.default.svc.cluster.local
Address 2: 10.244.2.23 myapp-1.myapp-svc.default.svc.cluster.local
/ #
```

Headless Service资源借助于SRV记录来引用真正提供服务的后端Pod资源的主机名称，进行指向包含Pod IP地址的记录条目。此外，由**StatefulSet**控制器管控的Pod资源终止后会由控制器自动进行重建，虽然其IP地址存在变化的可能性，但它的名称标识在重建后会保持不变。例如，在另一个终端中删除Pod资源**myapp-1**：

```
~]$ kubectl delete pods myapp-1
pod "myapp-1" deleted
```

删除完成后控制器将随之开始重建Pod资源，由下面的命令结果可知，其名称标识符的确未发生改变：

```
~]$ kubectl get pods -l app=myapp-pod
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-0	1/1	Running	0	8m
myapp-1	0/1	ContainerCreating	0	0s

Pod资源重建完成后，再次由此前创建的交互式测试终端尝试进行名称解析测试。由下面的命令结果可知，Pod资源的DNS标识亦未

发生改变，但其IP地址会指向重建后的Pod资源地址：

```
/ # nslookup myapp-1.myapp-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:        myapp-1.myapp-svc
Address 1: 10.244.3.33 myapp-1.myapp-svc.default.svc.cluster.local
/ #
```

因此，当客户端尝试向StatefulSet资源的Pod成员发出访问请求时，应该针对HeadlessService资源的CNAME（myapp-svc.default.svc.cluster.local）记录进行，它指向的SRV记录包含了当前处于就绪状态的Pod资源。当然，若在配置Pod模板时定义了Pod资源的liveness probe和readiness probe，考虑到名称标识固定不变，也可以让客户端直接向SRV资源记录（myapp-0.myapp-svc和myapp-1.myapp-svc）发出请求。

2.Pod资源的专有存储卷

前面的StatefulSet资源示例中，控制器通过volumeClaimTemplates为每个Pod副本自动创建并关联一个PVC对象，它们分别绑定了一个动态供给的PV对象：

```
~]$ kubectl get pvc -l app=myapp-pod -o \
    custom-columns=NAME:metadata.name,VOLUME:spec.volumeName,STATUS:status.phase
NAME          VOLUME                                     STATUS
myappdata-myapp-0  pvc-405cf708-2ddc-11e8-b246-000c29be4e28  Bound
myappdata-myapp-1  pvc-495f87c8-2ddc-11e8-b246-000c29be4e28  Bound
```

PVC存储卷由Pod资源中的容器挂载到了/usr/share/nginx/html目录，此为容器应用的Nginx进程默认的文档根路径。下面通过kubectl exec命令为每个Pod资源于此目录中生成一个测试页面，用于存储卷持久性测试：

```
~]$ for i in 0 1; do kubectl exec myapp-$i -- sh -c \
    'echo $(date), Hostname: $(hostname) > /usr/share/nginx/html/index.html';
done
```

接下来基于一个由cirros镜像启动的客户端Pod对象进行访问测试，这样便可通过其DNS名称引用每个Pod资源：

```
~]$ kubectl run -it --image cirros client --restart=Never --rm /bin/sh
If you don't see a command prompt, try pressing enter.
/ # curl myapp-0.myapp-svc
Fri Mar 23 07:00:46 UTC 2018, Hostname: myapp-0
/ # curl myapp-1.myapp-svc
Fri Mar 23 07:00:46 UTC 2018, Hostname: myapp-1
/ #
```

删除StatefulSet控制器的Pod资源，其存储卷并不会被删除，除非用户或管理员手动操作移除操作。因此，在另一个终端中删除Pod资源myapp-0，经由StatefulSet控制器重建后，它依然会关联到此前的PVC存储卷上，且此前数据依旧可用：

```
~]$ kubectl delete pods myapp-0
pod "myapp-0" deleted
```

另一个终端中监控到的删除及重建过程如下面的命令及其结果所示：

```
~]$ kubectl get pods -l app=myapp-pod -w
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-0	1/1	Running	0	1h
myapp-1	1/1	Running	0	1h
myapp-0	0/1	Terminating	0	1h
myapp-0	0/1	Pending	0	0s
myapp-0	0/1	Pending	0	0s
myapp-0	0/1	ContainerCreating	0	0s
myapp-0	1/1	Running	0	11s

而后通过测试用的Pod资源接口再次对其进行访问测试，由如下内容可知，存储卷是复用此前的那个：

```
/ # curl myapp-0.myapp-svc
Fri Mar 23 07:00:46 UTC 2018, Hostname: myapp-0
/ #
```

由此表明，重建的Pod资源被重新调度至哪个节点，此前的PVC资源就会被分配至哪个节点，这样就真正实现了数据的持久化。

9.3 StatefulSet资源扩缩容

StatefulSet资源的扩缩容与Deployment资源相似，即通过修改资源的副本数来改动其目标Pod资源数量。对StatefulSet资源来说，`kubectl scale`和`kubectl patch`命令均可实现此功能，也可以使用`kubectl edit`命令直接修改其副本数，或者在修改配置文件之后，由`kubectl apply`命令重新声明。

例如，下面的命令即能将myapp中的Pod副本数量扩展至6个：

```
~]$ kubectl scale statefulset myapp --replicas=6
statefulset.apps "myapp" scaled
```

StatefulSet资源的扩展过程与创建过程的Pod资源生成策略相同，默认为顺次进行，而且其名称中的序号也将以现有Pod资源的最后一个序号向后进行：

```
~]$ kubectl get pods -l app=myapp-pod -w
NAME          READY   STATUS             RESTARTS   AGE
myapp-0       1/1     Running            0           5m
myapp-1       1/1     Running            0           4m
myapp-2       0/1     ContainerCreating  0           10s
myapp-2       1/1     Running            0           25s
myapp-3       0/1     ContainerCreating  0           10s
...
```

与扩容操作相对，执行缩容操作只需要将其副本数量调低即可，例如，这里可使用`kubectl patch`命令将StatefulSet资源myapp的副本数量修补为3个：

```
~]$ kubectl patch statefulset myapp -p '{"spec":{"replicas":3}}'
statefulset.apps "myapp" patched
```

缩减规模时终止Pod资源的默认策略也以Pod顺序号逆序逐一进行，直到余下的数量满足目标为止：

```
~]$ kubectl get pods -l app=myapp-pod -w
NAME          READY   STATUS    RESTARTS   AGE
myapp-0       1/1     Running   0           7m
myapp-1       1/1     Running   0           6m
myapp-2       1/1     Running   0           2m
myapp-3       1/1     Running   0           2m
myapp-4       1/1     Running   0           1m
myapp-5       1/1     Running   0           1m
myapp-5       1/1     Terminating 0           1m
myapp-5       0/1     Terminating 0           1m
myapp-4       1/1     Terminating 0           2m
myapp-4       0/1     Terminating 0           2m
myapp-3       1/1     Terminating 0           2m
myapp-3       0/1     Terminating 0           2m
```

另外，终止Pod资源后，其存储卷并不会被删除，因此缩减规模后若再将其扩展回来，那么此前的数据依然可用，且Pod资源名称保持不变。

9.4 StatefulSet资源升级

自Kubernetes 1.7版本起，StatefulSet资源支持自动更新机制，其更新策略将由spec.updateStrategy字段定义，默认为RollingUpdate，即滚动更新。另一个可用策略为OnDelete，

即删除Pod资源重建以完成更新，这也是Kubernetes 1.6及之前版本唯一可用的更新策略。StatefulSet资源的更新机制可用于更新Pod资源中的容器镜像、标签、注解和系统资源配额等。

9.4.1 滚动更新

滚动更新StatefulSet控制器的Pod资源以逆序的形式从其最大索引编号的Pod资源逐一进行，它在终止一个Pod资源、更新资源并待其就绪后启动更新下一个资源，即索引号比当前号小1的Pod资源。对于主从复制类的集群应用来说，这样也能保证起主节点作用的Pod资源最后进行更新，确保兼容性。

StatefulSet的默认更新策略为滚动更新，通过“`kubectl get statefulset NAME`”命令中的输出可以获取相关的信息，myapp控制器的输出如下所示：

```
updateStrategy:
  rollingUpdate:
    partition: 0
  type: RollingUpdate
```

更新Pod中的容器镜像可以使用“`kubectl set image`”命令进行，例如下面的命令可将myapp控制器下的Pod资源镜像版本升级为“`ikubernetes/myapp: v6`”：

```
~]$ kubectl set image statefulset myapp myapp=ikubernetes/myapp:v6
statefulset.apps "myapp" image updated
```

更新操作过程，可在另一终端中使用如下命令进行监控，其逆序操作Pod资源的过程如下面的命令结果所示：

```
~]$ kubectl get pods -l app=myapp-pod -w
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-0	1/1	Running	0	11m
myapp-1	1/1	Running	0	11m
myapp-2	1/1	Running	0	6m
myapp-2	1/1	Terminating	0	7m
myapp-2	0/1	Terminating	0	7m
myapp-2	0/1	Pending	0	0s
myapp-2	0/1	ContainerCreating	0	0s
myapp-2	1/1	Running	0	11s
myapp-1	1/1	Terminating	0	11m
myapp-1	0/1	Terminating	0	11m
myapp-1	0/1	Pending	0	0s

myapp-1	0/1	ContainerCreating	0	2s
myapp-1	1/1	Running	0	18s
myapp-0	1/1	Terminating	0	12m
myapp-0	0/1	Terminating	0	12m
myapp-0	0/1	Pending	0	0s
myapp-0	0/1	ContainerCreating	0	0s
myapp-0	1/1	Running	0	11s

待更新完成后，获取每个Pod资源中的镜像文件版本即可验证其升级结果：

```
~]$ for i in 0 1 2; do kubectl get po myapp-$i \
  --template '{{range $i, $c := .spec.containers}}{{ $c.image}}{{end}}'; echo;
done
ikubernetes/myapp:v6
ikubernetes/myapp:v6
ikubernetes/myapp:v6
```

另外，用户也可以使用“`kubectl rollout status`”命令跟踪StatefulSet资源滚动更新过程中的状态信息。

9.4.2 暂存更新操作

当用户需要设定一个更新操作，但又不希望它立即执行时，可将更新操作予以“暂存”，待条件满足后再手动触发其执行更新。

StatefulSet资源的分区更新机制能够实现此项功能。在设定更新操作之前，将.spec.update-Strategy.rollingUpdate.partition字段的值设置为Pod资源的副本数量，即比Pod资源的最大索引号大1，这就意味着，所有的Pod资源都不会处于可直接更新的分区之内（如图9-4所示），那么于其后设定的更新操作也就不会真正执行，直到用户降低分区编号至现有Pod资源索引号范围之内。

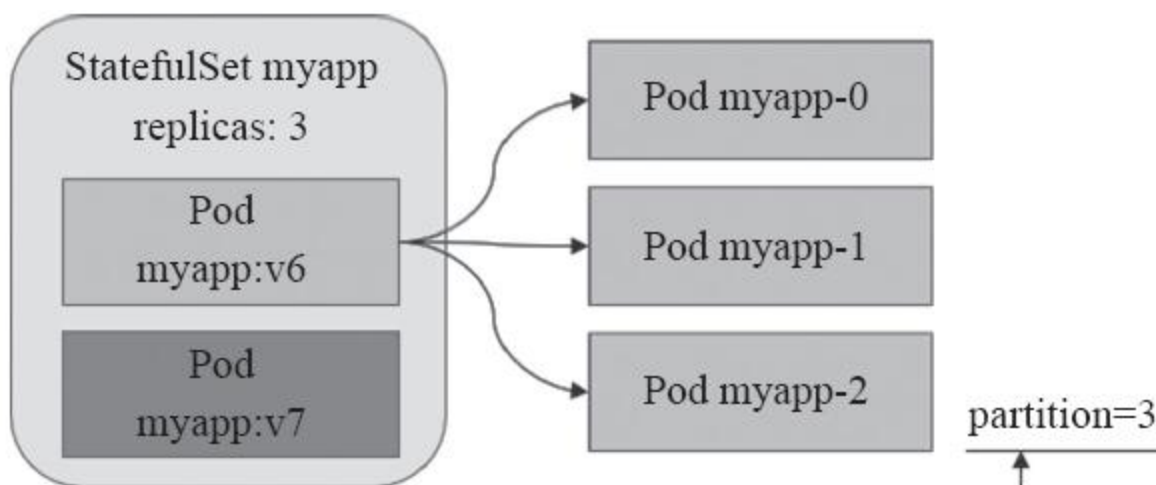


图9-4 暂存更新

下面测试滚动更新的暂存更新操作，首先将StatefulSet资源myapp的滚动更新分区值设定为3:

```
~]$ kubectl patch statefulset myapp \
  -p '{"spec":{"updateStrategy":{"rollingUpdate":{"partition":3}}}}'
statefulset.apps "myapp" patched
```

而后，将myapp控制器的Pod资源镜像版本更新为“ikubernetes/myapp: v7”:

```
~]$ kubectl set image statefulset myapp myapp=ikubernetes/myapp:v7
statefulset.apps "myapp" image updated
```

接着检测各Pod资源的镜像文件版本信息，可以发现其版本并未发生改变：

```
$ kubectl get pods -l app=myapp-pod \
-o custom-columns=NAME:metadata.name,IMAGE:spec.containers[0].image
NAME      IMAGE
myapp-0   ikubernetes/myapp:v6
myapp-1   ikubernetes/myapp:v6
myapp-2   ikubernetes/myapp:v6
```

此时，即便删除某Pod资源，它依然会基于旧的版本镜像进行重建。例如，下面首先删除了Pod对象myapp-1，随后待其重建操作启动后，再获取与其相关的镜像信息，结果依然显示了旧的版本：

```
~]$ kubectl delete pods myapp-1
pod "myapp-1" deleted
~]$ kubectl get pods myapp-1 \
-o custom-columns=NAME:metadata.name,IMAGE:spec.containers[0].image
NAME      IMAGE
myapp-1   ikubernetes/myapp:v6
```

由此可见，[暂存状态的更新操作](#)对所有的Pod资源均不产生影响。

9.4.3 金丝雀部署

将处于暂存状态的更新操作的`partition`定位于Pod资源的最大索引号，即可放出一只金丝雀，由其测试第一轮更新操作，在确认无误后通过修改`partition`属性的值更新其他的Pod对象是一种更为稳妥的更新操作。

将9.4.2节暂停的更新StatefulSet控制器myapp资源的分区号设置为Pod资源的最大索引号2，将会触发myapp-2的更新操作：

```
~]$ kubectl patch statefulset myapp -p '{"spec":{"updateStrategy":  
{"rollingUpdate":{"partition":2}}}}'  
statefulset.apps "myapp" patched
```

其执行结果如图9-5所示。

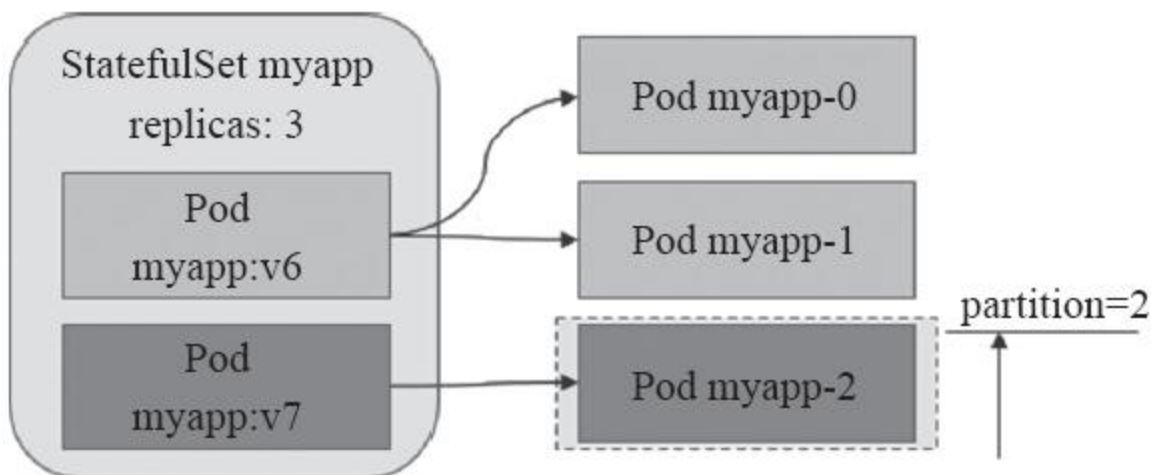


图9-5 金丝雀发布

待其更新完成后，检测所有Pod资源使用的镜像文件版本，对比下面命令的输出，可以看出其更新操作再次暂停于myapp-2的更新操作完成之后：

```
~]$ kubectl get pods -l app=myapp-pod \
-o custom-columns=NAME:metadata.name,IMAGE:spec.containers[0].image
NAME      IMAGE
```

myapp-0	ikubernetes/myapp:v6
myapp-1	ikubernetes/myapp:v6
myapp-2	ikubernetes/myapp:v7

此时，位于非更新分区内的其他**Pod**资源仍不会被更新到新的镜像版本，哪怕它们被删除后重建亦是如此。

9.4.4 分段更新

金丝雀安然度过测试阶段之后，用户便可启动后续其他Pod资源的更新操作。在待更新的Pod资源数量较少的情况下，直接将partition属性的值设置为0，它将逆序完成后续所有Pod资源的更新。而当待更新的Pod资源较多时，用户也可以将Pod资源以线性或指数级增长的方式来分阶段完成更新操作，操作过程无非是分步更新partition属性值，例如，将myapp控制器的分区号码依次设置为1、0以完成剩余Pod资源的线性分步更新，如图9-6所示。

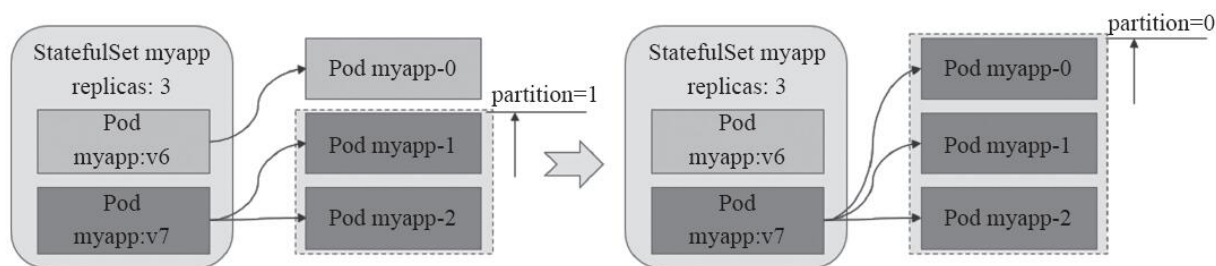


图9-6 分阶段更新

待更新全部完成后，用户便可根据需求筹划下一轮的更新操作。

9.4.5 其他话题

同其他类型的Pod控制器资源类似，**StatefulSet**也支持级联或非级联的删除操作。默认的删除类型为级联删除，即同时删除**StatefulSet**和相关的Pod资源。若要执行非级联删除，为删除命令使用“`--cascade=false`”选项即可。

另外，**StatefulSet**控制器管理Pod资源的策略除了默认的**OrderedReady**（顺次创建及逆序删除）之外，还支持并行的创建和删除操作，即同时创建所有的Pod资源以及同时删除所有的Pod资源，完成这一点，只需要将**spec.podManagementPolicy**字段的值设置为**Parallel**即可，不过对于有角色之分的分布式应用来说，为了保证数据安全可靠，建议使用默认策略，除非数据完整性是可以不用考虑在内的因素。

另外，不同的有状态应用的运维操作过程差别巨大，因此**StatefulSet**控制器本身几乎无法为此种类型的应用提供完善的通用管理控制机制，现实中的各种有状态应用通常是使用专用的自定义控制器专门封装特定的运维操作流程，这些自定义控制器有时也被统一称为**Operator**，这些内容将在后面的章节介绍。

9.5 案例：etcd集群

Kubernetes的所有对象都需要持久化存储于etcd存储系统中，以确保系统重启或故障恢复后能将它们予以还原。

etcd是一个分布式键值数据存储系统，具有可靠、快速、强一致性等特性，它通过分布式锁、leader选举和写屏障（write barriers）来实现可靠的分布式协作。因此，Kubernetes系统管理员应该将etcd部署为集群工作模型，以实现其服务高可用，并提供更好的性能表现。

etcd将键存储于层级组织的键空间中，这使得它看起来非常类似于文件系统的倒置树状结构，于是，它的每个键要么是一个包含有其他键的目录，要么是一个含有数据的常规键。Kubernetes将其所有数据存储于etcd的/registry目录中，虽然API Server是以JSON格式组织数据资源对象，但对于底层使用etcd存储系统的场景来说，它们可类比为存储于文件系统内的JSON文件中。另外，Kubernetes的系统组件中，仅API Server直接与etcd进行通信，其他所有组件的数据读写操作都要经由API Server进行，从而确保了数据的高度一致性。

在分布式模型中，etcd采用raft算法，实现了分布式系统数据的可用性和一致性。若发生网络分区或节点故障，则raft算法就会通过quorum机制处理集群状态转移，其中成员数量大于半数的一方可继续工作，若leader存在，则它们可继续提供读写服务，否则就要选举出新的leader，并且在此期间写操作是被禁止的，而成员数量少于半数的节点将停止任何的写入操作。本节将描述如何于Kubernetes集群中基于StatefulSet控制器托管部署一个分布式的etcd集群，它只是一个部署示例，并不会取代Kubernetes系统现有的etcd。

9.5.1 创建Service资源

StatefulSet资源依赖于Headless Service为各Pod资源提供名称解析服务，其他Pod资源可直接使用DNS名称来获取相关的服务，如etcd-0.etcd、etcd-1.etcd等，下面的资源清单（etcd-services.yaml）中定义的第一个资源etcd就是此类的Service资源。第二个名为etcd-client的Service资源是一个正常的Service，它拥有ClusterIP，并可通过NodePort向Kubernetes集群外部的etcd客户端提供服务：

```
apiVersion: v1
kind: Service
metadata:
  name: etcd
  annotations:
    # Create endpoints also if the related pod isn't ready
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
spec:
  ports:
    - port: 2379
      name: client
    - port: 2380
      name: peer
  clusterIP: None
  selector:
    app: etcd-member
---
apiVersion: v1
kind: Service
metadata:
  name: etcd-client
spec:
  ports:
    - name: etcd-client
      port: 2379
      protocol: TCP
      targetPort: 2379
  selector:
    app: etcd-member
  type: NodePort
```

首先创建上述两个Service对象：

```
~]$ kubectl apply -f etcd-services.yaml
service "etcd-svc" created
service "etcd-client" created
```

而后，可于**default**名称空间中看到如下两个**Service**资源记录：

~]\$ kubectl get svc -l app=etcd					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
etcd	ClusterIP	None	<none>	2379/TCP, 2380/TCP	5s
etcd-client	NodePort	10.244.60.53	<none>	2379:30290/TCP	5s

由上面输出的信息可以看出，**etcd-client**通过**10.99.175.53**向客户端提供服务，它是一个标准类型的**Service**资源，类型为**NodePort**，可通过各工作节点的**30290**端口将服务暴露到集群外部。服务类型的资源创建完成后，接下来便可创建**StatefulSet**控制器，构建分布式**etcd**集群。

9.5.2 etcd StatefulSet

etcd依赖于持久存储设备保存数据，这里为其配置了使用自定义的、具有动态PV供给功能的gluster-dynamic存储类为各Pod资源的PVC提供存储后端，大小可由用户按需求进行定义。另外，etcd镜像文件的版本也可由用户修改为与实际需求匹配的版本，如3.3.1等，不过StatefulSet本就支持自动更新机制，用户也可以于必要时启动更新机制切换其镜像版本。下面是定义在etcd-statefulset.yaml文件中的配置示例，它定义了一个名为etcd的StatefulSet资源：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: etcd
  labels:
    app: etcd
spec:
  serviceName: etcd
  # changing replicas value will require a manual etcdctl member remove/add
  # command (remove before decreasing and add after increasing)
  replicas: 3
  selector:
    matchLabels:
      app: etcd-member
  template:
    metadata:
      name: etcd
      labels:
        app: etcd-member
    spec:
      containers:
        - name: etcd
          image: "quay.io/coreos/etcd:v3.2.16"
          ports:
            - containerPort: 2379
              name: client
            - containerPort: 2380
              name: peer
          env:
            - name: CLUSTER_SIZE
              value: "3"
            - name: SET_NAME
              value: "etcd"
          volumeMounts:
            - name: data
              mountPath: /var/run/etcd
          command:
            - "/bin/sh"
            - "-ecx"
            - |
              IP=$(hostname -i)
```

```

PEERS=""
for i in $(seq 0 $(( ${CLUSTER_SIZE} - 1 ))); do
    PEERS="${PEERS}${PEERS:+,}${SET_NAME}-${i}=http://${SET_NAME}-${i}.${SET_NAME}:2380"
done
# start etcd. If cluster is already initialized the `--initial-*`
# options will be ignored.
exec etcd --name ${HOSTNAME} \
    --listen-peer-urls http://${IP}:2380 \
    --listen-client-urls http://${IP}:2379,http://127.0.0.1:2379 \
    --advertise-client-urls http://${HOSTNAME}.${SET_NAME}:2379 \
    --initial-advertise-peer-urls http://${HOSTNAME}.${SET_NAME}:2380 \
    --initial-cluster-token etcd-cluster-1 \
    --initial-cluster ${PEERS} \
    --initial-cluster-state new \
    --data-dir /var/run/etcd/default.etcd
volumeClaimTemplates:
- metadata:
    name: data
  spec:
    storageClassName: gluster-dynamic
    accessModes:
      - "ReadWriteOnce"
    resources:
      requests:
        storage: 1Gi

```

首先，将上述资源定义创建于集群中。当依赖的镜像文件不存在时，下载镜像的过程会导致创建时长略长，此处需要耐心等待：

```

~]$ kubectl apply -f etcd-statefulset.yaml
statefulset.apps "etcd" created

```

在另一终端，可使用下列命令监控各Pod资源的创建过程，直到它们都转为正常的运行状态为止：

```

~]$ kubectl get pods -l app=etcd-member -w

```

NAME	READY	STATUS	RESTARTS	AGE
etcd-0	0/1	Pending	0	0s
etcd-0	0/1	Pending	0	0s
etcd-0	0/1	ContainerCreating	0	0s
etcd-0	1/1	Running	0	11s
etcd-1	0/1	Pending	0	0s
etcd-1	0/1	Pending	0	0s
etcd-1	0/1	ContainerCreating	0	0s
etcd-1	1/1	Running	0	11s
etcd-2	0/1	Pending	0	0s
etcd-2	0/1	Pending	0	0s
etcd-2	0/1	ContainerCreating	0	1s
etcd-2	1/1	Running	0	15s

而后，检测9.5.1节中创建的Service资源etcd-svc和etcd-client是否已经正常关联到相关的Pod资源。下面的命令结果表示它们都已经通过标签选择器关联到了由Pod资源生成的Endpoints资源之上，这些Pod资源是由StatefulSet控制器etcd创建而成的：

```
~]$ kubectl get endpoints -l app=etcd
```

NAME	ENDPOINTS	AGE
etcd	10.244.1.50:2380,10.244.2.50:2380,10.244.3.57:2380 + 3 more...	1m
etcd-client	10.244.1.50:2379,10.244.2.50:2379,10.244.3.57:2379	1m

对比由etcd控制器创建的Pod资源的相关信息可知，上面的Endpoints中的IP地址均属于由etcd控制创建的Pod资源：

```
~]$ kubectl get pods -l app=etcd-member -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
etcd-0	1/1	Running	0	1m	10.244.3.57	node03.ilinux.io
etcd-1	1/1	Running	0	1m	10.244.1.50	node01.ilinux.io
etcd-2	1/1	Running	0	1m	10.244.2.50	node02.ilinux.io

使用etcd的客户端工具etcdctl检测集群的健康状态：

```
~]$ kubectl exec etcd-0 -- etcdctl cluster-health
```

```
member 2e80f96756a54ca9 is healthy: got healthy result from http://etcd-0.etcd:2379
member 7fd61f3f79d97779 is healthy: got healthy result from http://etcd-1.etcd:2379
member b429c86e3cd4e077 is healthy: got healthy result from http://etcd-2.etcd:2379
cluster is healthy
```

至此，一个由三节点（Pod资源）组成的etcd集群已经正常运行起来了，客户端可从各工作节点的30290端口在Kubernetes集群外部进行访问，而各Pod资源既可以直接向Service资源etcd-client的名称或10.106.60.53（ClusterIP）发出访问请求，也可以向Headless Service资源的名称etcd进行请求。

需要注意的是，前面的配置文件中提供的StatefulSet资源配置要求进行规模变动时，需要手动对etcd集群执行命令以完成节点的添加或删除，而且缩容时要先移除节点再缩减副本数量，扩容时要先提升副本数量再添加节点，因此它的主要目标里不包含应用规模的自由变

动，但对于应用升级的支持完好。例如，将镜像文件版本升级至v3.2.17的版本：

```
~]$ kubectl set image statefulset etcd etcd=quay.io/coreos/etcd:v3.2.17
statefulset.apps "etcd" image updated
```

而后使用“`kubectl rollout status`”命令监控其滚动升级过程中的状态变动：

```
~]$ kubectl rollout status statefulset etcd
Waiting for 1 pods to be ready...
Waiting for partitioned roll out to finish: 1 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for partitioned roll out to finish: 2 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
partitioned roll out complete: 3 new pods have been updated...
```

升级完成后，可于相关的任一Pod资源中运行`etcdctl`命令查看应用程序的版本是否已经升级完成：

```
~]$ kubectl exec etcd-0 -- etcdctl -v
etcdctl version: 3.2.17
API version: 2
```

由上面命令结果可见，应用程序已经成功升级至3.2.17的版本。滚动升级过程中，处于更新过程的Pod资源对象无法正常使用，不过，客户端是通过Service资源提供的入口而非Pod资源本身的标识（如名称或IP地址）来进行访问，因此并不会出现服务不可用问题。

9.6 本章小结

本章着重讲解了有状态应用的Pod资源控制器StatefulSet，有状态应用相较于无状态应用来说，在管理上有着特有的复杂之处，甚至不同的有状态应用的管理方式各不相同，在部署时需要予以精心组织。

- StatefulSet依赖于Headless Service资源为其Pod资源创建DNS资源记录。

- 每个Pod资源均拥有固定且唯一的名称，并且需要由DNS服务解析。

- Pod资源中的应用需要依赖于PVC和PV持久保存其状态数据。

- 支持扩容和缩容，但具体的实现机制依赖于应用本身。

- 支持自动更新，默认的更新策略为滚动更新机制。

第10章 认证、授权与准入控制

在任何将资源或服务提供给有限使用者的系统上，认证和授权都是两个必不可少的功能，认证用于身份鉴别，而授权则实现权限分派。**Kubernetes**以插件化的方式实现了这两种功能，且分别存在多种可用的插件。另外，它还支持准入控制机制，用于补充授权机制以实现更精细的访问控制功能。本章主要讲解**Kubernetes**系统常用的认证、授权及准入控制机制。

10.1 访问控制概述

API Server作为Kubernetes集群系统的网关，是访问及管理资源对象的唯一入口，余下所有需要访问集群资源的组件，包括kube-controller-manager、kube-scheduler、kubelet和kube-proxy等集群基础组件、CoreDNS等集群的附加组件以及此前使用的kubectl命令等都要经由此网关进行集群访问和管理。这些客户端均要经由API Server访问或改变集群状态并完成数据存储，并由它对每一次的访问请求进行合法性检验，包括用户身份鉴别、操作权限验证以及操作是否符合全局规范的约束等。所有检查均正常完成且对象配置信息合法性检验无误之后才能访问或存入数据于后端存储系统etcd中，如图10-1所示。

客户端认证操作由API Server配置的一到多个认证插件完成。收到请求后，API Server依次调用为其配置的认证插件来认证客户端身份，直到其中一个插件可以识别出请求者的身份为止。授权操作由一到多个授权插件进行，它负责确定那些通过认证的用户是否有权限执行其发出的资源操作请求，如创建、读取、删除或修改指定的对象等。随后，通过授权检测的用户所请求的修改相关的操作还要经由一到多个准入控制插件的遍历检测，例如使用默认值补足要创建的目标资源对象中未定义的各字段、检查目标Namespace资源对象是否存在、检查是否违反系统资源限制，等等，而其中任何的检查失败都可能会导致写入操作失败。

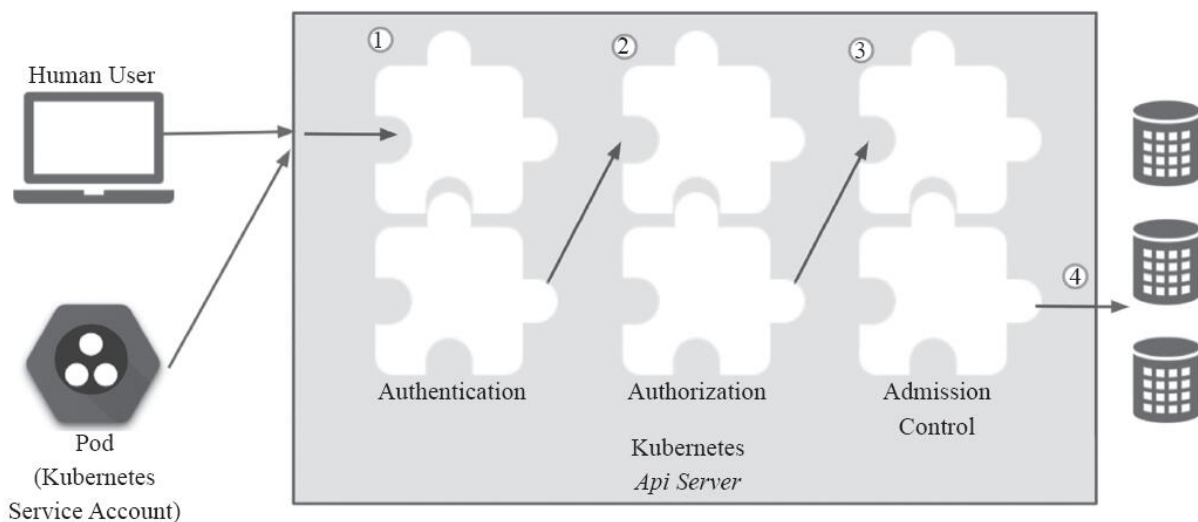


图10-1 用户账号、服务账号、认证、授权和准入控制

10.1.1 用户账户与用户组

Kubernetes并不会存储由认证插件从客户端请求中提取出的用户及所属组的信息，它们仅仅用于检验用户是否有权限执行其所请求的操作。客户端访问API服务的途径通常有三种：**kubectl**、客户端库或者直接使用**REST**接口进行请求，而可以执行此类请求的主体也被Kubernetes分为两类：现实中的“人”和Pod对象，它们的用户身份分别对应于常规用户（**User Account**）和服务账号（**Service Account**）。

- User Account**（用户账号）：一般是指由独立于Kubernetes之外的其他服务管理的用户账号，例如由管理员分发的密钥、**Keystone**一类的用户存储（账号库）、甚至是包含有用户名和密码列表的文件等。Kubernetes中不存在表示此类用户账号的对象，因此不能被直接添加进Kubernetes系统中。

- Service Account**（服务账号）：是指由Kubernetes API管理的账号，用于为Pod之中的服务进程在访问Kubernetes API时提供身份标识（**identity**）。**Service Account**通常要绑定于特定的名称空间，它们由API Server创建，或者通过API调用手动创建，附带着一组存储为**Secret**的用于访问API Server的凭据。

User Account通常用于复杂的业务逻辑管控，它作用于系统全局，故其名称必须全局唯一。相比较来说，**Service Account**隶属于名称空间，仅用于实现某些特定的操作任务，因此要轻量得多。这两类账号都可以隶属于一个或多个用户组。用户组只是用户账号的逻辑集合，它本身并没有操作权限，但附加于组上的权限可由其内部的所有用户继承，以实现高效的授权管理机制。Kubernetes有着以下几个内建的用于特殊目的的组。

- system: unauthenticated**：未能通过任何一个授权插件检验的账号，即未通过认证测试的用户所属的组。

- system: authenticated**：认证成功后的用户自动加入的一个组，用于快捷引用所有正常通过认证的用户账号。

·system: serviceaccounts: 当前系统上的所有Service Account对象。

·system: serviceaccounts: <namespace>: 特定名称空间内所有的Service Account对象。

API请求要么与普通用户或服务账户进行绑定，要么被视为匿名请求。这意味着群集内部或外部的每个进程，包括由人类用户使用的kubect1，到节点上的kubelet，再到控制平面的成员组件，必须在向API服务器发出请求时进行身份验证，否则即被视为匿名用户。

10.1.2 认证、授权与准入控制基础

API Server处理请求的过程中，认证插件负责鉴定用户身份，授权插件用于操作权限许可鉴别，而准入控制则用于在资源对象的创建、删除、更新或连接（**proxy**）操作时实现更精细的许可检查，其相互间的作用关系如图10-1所示。

Kubernetes使用身份验证插件对API请求进行身份验证，支持的认证方式包括客户端证书、承载令牌（**bearer tokens**）、身份验证代理（**authenticating proxy**）或HTTP basic认证等。API Server接收到访问请求时，它将调用认证插件尝试将以下属性与访问请求相关联。

- Username**: 用户名，如kubernetes-admin等。
- UID**: 用户的数字标签符，用于确保用户身份的唯一性。
- Groups**: 用户所属的组，用于权限指派和继承。
- Extra**: 键值数据类型的字符串，用于提供认证时需要用到的额外信息。

API Server支持同时启用多种认证机制，但至少应该分别为**Service Account**和**User Account**各自启用一个认证插件。同时启用多种认证机制时，认证过程会以串行的方式进行，直到一种认证机制成功完成即结束。若认证失败，则服务器会响应**401**状态码，反之，请求者就会被识别为某个具体的用户（以其用户名进行标识），并且随后的操作都将以此用户身份来进行。

具体来说，API Server支持以下几种具体的认证方式，其中所有的令牌认证机制通常被统称为承载令牌认证。

- 1) **X509客户端证书认证**: 客户端在请求报文中携带X509格式的数字证书用于认证，认证通过后，证书中的主体标识（**Subject**）将被识别为用户标识，其中的CN（**Common Name**）字段是用户名，O（**Organization**）是用户所属的组，

如“/CN=ilinux/O=opmasters/O=admin”中，用户名为ilinux，其属于opmasters和admin两个组。

2) 静态令牌文件 (Static Token File)：即保存着令牌信息的文件，由kube-apiserver的命令行选项--token-auth-file加载，且服务器启动后不可更改；HTTP客户端也能使用承载令牌进行身份验证，将令牌进行编码后，通过请求报文中的Authorization首部承载传递给API Server即可。

3) 引导令牌 (Bootstrap Tokens)：一种动态管理承载令牌进行身份认证的方式，常用于简化新建Kubernetes集群的节点认证过程，需要通过--experimental-bootstrap-token-auth选项启用；有新的工作节点首次加入时，Master使用引导令牌确认节点身份的合法性之后自动为其签署数字证书以用于后续的安全通信，使用kubeadm join命令将节点加入kubeadm初始化的集群时使用的即是这种认证方式；这些令牌作为Secrets存储在kube-system命名空间中时，可以动态管理和创建它们，而Controller Manager包含一个TokenCleaner控制器，用于删除过期的引导令牌。

4) 静态密码文件：用户名和密码等令牌以明文格式存储的CSV格式文件，由kube-apiserver使用--basic-auth-file选项进行加载；客户端在HTTP basic认证中将认证用户的用户名和密码编码后以承载令牌的方式进行认证。

5) 服务账户令牌：由kube-apiserver自动启用，并可使用可选选项加载--service-account-key-file验证承载令牌的密钥，省略时将使用kube-apiserver自己的证书匹配的私钥文件；Service Account通常由API Server自动创建，并通过ServiceAccount准入控制器将其注入Pod对象，包括Service Account上的承载令牌，容器中的应用程序请求API Server的服务时将以此完成身份认证。

6) OpenID连接令牌：OAuth2的一种认证风格，由Azure AD、Salesforce和Google等OAuth2服务商所支持，协议的主要扩展是返回的附加字段，其中的访问令牌也称为ID令牌；它属于JSON Web令牌 (JWT) 类型，有着服务器签名过的常用字段，如email等；kube-apiserver启用这种认证功能的相关选项较多。

7) Webhook令牌: HTTP身份验证允许将服务器的URL注册为Webhook, 并接收带有承载令牌的POST请求进行身份认证; 客户端使用kubeconfig格式的配置文件, 在文件中, “users”指的是API服务器Webhook, “clusters”指的是API Server。

8) 认证代理: API Server支持从请求首部的值中识别用户, 如X-Remote-User首部, 它旨在与身份验证代理服务相结合, 并由该代理设置相应的请求首部。

9) Keystone密码: 借助于外部的Keystone服务器进行身份认证。

10) 匿名请求: 未被任何验证机制明确拒绝的用户即为匿名用户, 其会被自动标识为用户名system: anonymous, 并隶属于system: unauthenticated用户组; 在API Server启用了除AlwaysAllow以外的认证机制时, 匿名用户处于启用状态, 不过, 管理员可通过--anonymous-auth=false选项将其禁用。

另外, API Server还允许用户通过模拟 (impersonation) 首部来冒充另一个用户, 这些请求可以以手动的方式覆盖请求中用于身份验证的用户信息。例如, 管理员可以使用此功能临时模拟其他用户来查看请求是否被拒绝进行授权策略调试。

API Server是一种REST API, 除了身份认证信息之外, 一个请求报文还需要提供操作方法及其目标对象, 如针对某Pod资源对象进行的创建、查看、修改或删除操作等, 具体来说, 请求报文包含如下信息。

- API: 用于定义请求的目标是否为一个API资源。

- Request path: 请求的非资源型路径, 如/api或/healthz。

- API group: 要访问的API组, 仅对资源型请求有效; 默认为“core API group”。

- Namespace: 目标资源所属的名称空间, 仅对隶属于名称空间类型的资源有效。

- API request verb: API请求类的操作，即资源型请求（对资源执行的操作），包括get、list、create、update、patch、watch、proxy、redirect、delete和deletecollection等。

- HTTP request verb: HTTP请求类的操作，即非资源型请求要执行的操作，如get、post、put和delete。

- Resource: 请求的目标资源的ID或名称。

- Subresource: 请求的子资源。

为了核验用户的操作许可，成功通过身份认证后的操作请求还需要转交给授权插件进行许可权限检查，以确保其拥有执行相应的操作的许可。API Server主要支持使用四类内建的授权插件来定义用户的操作权限。

- Node: 基于Pod资源的目标调度节点来实现的对kubelet的访问控制。

- ABAC: attribute-based access control，基于属性的访问控制。

- RBAC: role-based access control，基于角色的访问控制。

- Webhook: 基于HTTP回调机制通过外部REST服务检查确认用户授权的访问控制。

另外，还有AlwaysDeny和AlwaysAllow两个特殊的授权插件，其中AlwaysDeny（总是拒绝）仅用于测试，而AlwaysAllow（总是允许）则用于不期望进行授权检查时直接于授权检查阶段放行的所有操作请求。启动API Server时，“--authorization-mode”选项用于定义要启用的授权机制，多个选项彼此之间以逗号分隔。

而准入控制器（Admission Controller）则用于在客户端请求经过身份验证和授权检查之后，但在对象持久化存储etcd之前拦截请求，用于实现在资源的创建、更新和删除操作期间强制执行对象的语义验证等功能，读取资源信息的操作请求不会经由准入控制器的检查。API Server内置了许多准入控制器，常用的包含如下几种。不过，其中的个别控制器仅在较新版本的Kubernetes中才受支持。

- 1) **AlwaysAdmit**: 允许所有请求。
- 2) **AlwaysDeny**: 拒绝所有请求，仅应该用于测试。
- 3) **AlwaysPullImages**: 总是下载镜像，即每次创建Pod对象之前都要去下载镜像，常用于多租户环境中以确保私有镜像仅能够被拥有权限的用户使用。
- 4) **NamespaceLifecycle**: 拒绝于不存在的名称空间中创建资源，而删除名称空间将会级联删除其下的所有其他资源。
- 5) **LimitRanger**: 可用资源范围界定，用于监控对设置了LimitRange的对象所发出的所有请求，以确保其资源请求不会超限。
- 6) **ServiceAccount**: 用于实现Service Account管控机制的自动化，实现创建Pod对象时自动为其附加相关的Service Account对象。
- 7) **PersistentVolumeLabel**: 为那些由云计算服务商提供的PV自动附加region或zone标签，以确保这些存储卷能够正确关联且仅能关联到所属的region或zone。
- 8) **DefaultStorageClass**: 监控所有创建PVC对象的请求，以保证那些没有附加任何专用StorageClass的请求会自动设定一个默认值。
- 9) **ResourceQuota**: 用于对名称空间设置可用资源的上限，并确保在其中创建的任何设置了资源限额的对象都不会超出名称空间的资源配额。
- 10) **DefaultTolerationSeconds**: 如果Pod对象上不存在污点宽容期限，则为它们设置默认的宽容期，以宽容“notready: NoExecute”和“unreachable: NoExecute”类的污点5分钟时间。
- 11) **ValidatingAdmissionWebhook**: 并行调用匹配当前请求的所有验证类的Webhook，任何一个校验失败，请求即失败。
- 12) **MutatingAdmissionWebhook**: 串行调用匹配当前请求的所有变异类的Webhook，每个调用都可能会更改对象。

早期的准入控制器代码需要由管理员编译进kube-apiserver中才能使用，其实现方式缺乏灵活性。于是，Kubernetes自1.7版本引入了Initializers和External Admission Webhooks来尝试突破此限制，而且自1.9版本起，External Admission Webhooks被分为MutatingAdmission-Webhook和ValidatingAdmissionWebhook两种类型，分别用于在API中执行对象配置的变异和验证操作。检查期间，只有那些顺利通过所有准入控制器检查的资源操作请求的结果才能保存到etcd中，实现持久存储，换句话讲，任何一个准入控制器的拒绝都将导致请求失败。

10.2 服务账户管理与应用

运行过程中，Pod资源里的容器进程在某些场景中需要调用Kubernetes API或者其他类型的服务，而这些服务通常需要认证客户端身份，如调度器、Pod控制器或节点控制器，甚至是获取启动容器的镜像访问的私有Registry服务等。服务账户就是用于让Pod对象内的容器进程访问其他服务时提供身份认证信息的账户。一个Service Account资源一般由用户名及相关的Secret对象组成。

10.2.1 Service Account自动化

细心的读者或许已经注意到，此前创建的每个Pod资源都自动关联了一个存储卷，并由其容器挂载至/var/run/secrets/kubernetes.io/serviceaccount目录，例如，下面由“`kubectl describe pod`”命令显示的某Pod对象描述信息的片断：

```
Containers:
.....
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-bq6zc (ro)
.....
Volumes:
  default-token-bq6zc:
    Type:          Secret (a volume populated by a Secret)
    SecretName:    default-token-bq6zc
    Optional:      false
```

挂载点目录中通常存在三个文件：`ca.crt`、`namespace`和`token`，其中，`token`文件保存了Service Account的认证token，容器中的进程使用它向API Server发起连接请求，进而由认证插件完成用户认证并将其用户名传递给授权插件。

每个Pod对象都只有一个服务账户，若创建Pod资源时未予以明确指定，则名为ServiceAccount的准入控制器会为其自动附加当前名称空间中默认的服务账户，其名称通常为`default`。下面的命令显示了`default`这个服务账户的详细信息：

```
~]$ kubectl describe serviceaccount default
Name:          default
.....
Mountable secrets:  default-token-bq6zc
Tokens:           default-token-bq6zc
Events:           <none>
```

Kubernetes系统通过三个独立的组件间的相互协作来实现服务账户的自动化，三个组件具体为：Service Account准入控制器、令牌控制器（token controller）和Service Account账户控制器。Service Account控制器负责为名称空间管理相应的资源，并确保每个名称空间

中都存在一个名为“default”的Service Account对象。Service Account准入控制器是API Server的一部分，负责在创建或更新Pod时对其按需进行Service Account对象相关信息的修改，这包括如下操作。

- 若Pod没有明确定义使用的ServiceAccount对象，则将其设置为“default”。

- 确保Pod明确引用的ServiceAccount已存在，否则请求将被拒绝。

- 若Pod对象中不包含ImagePullSecerts，则把Service Account的ImagePullSecrets添加于其上。

- 为带有访问API的令牌的Pod添加一个存储卷。

- 为Pod对象中的每个容器添加一个volumeMounts，挂载至/var/run/secrets/kubernetes.io/serviceaccount。

令牌控制器是controller-manager的子组件，工作于异步模式。其负责完成的任务具体如下。

- 监控Service Account的创建操作，并为其添加用于访问API的Secret对象。

- 监控Service Account的删除操作，并删除其相关的所有Service Account令牌密钥。

- 监控Secret对象的添加操作，确保其引用的Service Account已存在，并在必要时为Secret对象添加认证令牌。

- 监控Secret对象的删除操作，以确保删除每个Service Account中对此Secret的引用。

需要注意的是，为确保完整性等，必须为kube-controller-manager使用“--service-account-private-key-file”选项指定一个私钥文件，以用于对生成的服务账户令牌进行签名，此私钥文件必须是pem格式。类似地，还要为kube-apiserver使用“--service-account-key-file”指定与前面的私钥配对儿的公钥文件，以用于在认证期间对认证令牌进行校验。

10.2.2 创建服务账户

Service Account是Kubernetes API上的一种资源类型，它隶属于名称空间，用于让Pod对象内部的应用程序在与API Server通信时完成身份认证。事实上，每个名称空间都有一个名为default的默认资源对象，如下面的命令及其结果所示，其可用于让Pod对象有权限读取同一名称空间中的其他资源对象的元数据信息。需要赋予Pod对象更多操作权限时，则应该由用户按需创建自定义的**Service Account**资源：

```
~]$ kubectl get serviceaccounts --all-namespaces
```

NAMESPACE	NAME	SECRETS	AGE
default	default	1	2h
kube-public	default	1	2h
kube-system	default	1	2h

每个Pod对象均可附加其所属名称空间中的一个**Service Account**资源，且只能附加一个。不过，一个**Service Account**资源可由其所属名称空间中的多个Pod对象共享使用。创建Pod资源时，用户可使用“**spec.serviceAccountName**”属性直接指定要使用的**Service Account**对象，或者省略此字段而由其自动附加当前名称空间中默认的**Service Account**（**default**）。

可使用资源配置文件创建**Service Account**资源，也可直接使用“**kubectl create servic-eaccount**”命令进行创建，提供认证令牌的**Secret**对象会由命令自动创建完成。下面的配置清单是一个**ServiceAccount**资源示例，它仅指定了创建名为**sa-demo**的服务账户，其余的信息则交由系统自动生成：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-demo
  namespace: default
```

创建Pod对象时，可为Pod对象指定使用自定义的服务账户，从而实现自主控制Pod对象资源的访问权限。Pod向API Server发出请求时，其携带的认证令牌在通过认证后将由授权插件来判定相关的

Service Account是否有权限访问其所请求的资源。Kubernetes支持多种授权插件，由管理员负责选定及配置，**RBAC**是目前较为主流的选择。

10.2.3 调用imagePullSecret资源对象

ServiceAccount资源还可以基于**spec.imagePullSecret**字段附带一个由下载镜像专用的**Secret**资源组成的列表，用于在进行容器创建时，从某私有镜像仓库下载镜像文件之前进行服务认证。下面的示例定义了一个有着从本地私有镜像仓库**harbor**中下载镜像文件时用于认证的**Secret**对象信息的**ServiceAccount**:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: image-download-sa
imagePullSecrets:
- name: local-harbor-secret
```

其中，**local-harbor-secret**是**docker-registry**类型的**Secret**对象，由用户提前手动创建，它可以通过键值数据提供**docker**仓库服务器的地址、接入服务器的用户名、密码及用户的电子邮件等信息。认证通过后，引用了此**ServiceAccount**的**Pod**资源即可从指定的镜像仓库中下载由**image**字段指定的镜像文件。

10.3 X.509数字证书认证

Kubernetes支持的HTTPS客户端证书认证、token认证及HTTP basic认证几种认证方式中，基于SSL/TLS协议的客户端证书认证以其安全性高且易于实现等特性，而成为主要使用的认证方式之一。

SSL/TLS最常见的使用场景是将X.509证书与服务器端相关联，但不为客户端使用证书。这意味着客户端可以验证服务端的身份，但服务端无法验证客户端的身份（至少不能通过SSL/TLS协议进行）。这很容易理解，毕竟SSL/TLS安全性最初是为互联网应用开发，保护客户端反而是高优先级的需求，例如，在需要连接到银行的网站时需要确保该网站是真实的等，如图10-2所示。

此外，验证客户端时，使用其他机制（如HTTP基本认证）通常会更容易些，而且这些机制不会产生生成和分发X.509证书的高昂维护开销。不过，安全性要求较高的场景中，使用组织私有的证书分发系统也一样能够使用数字证书进行客户端认证。图10-3展示了服务端与客户端的双向认证机制。

服务端与客户端互相认证的场景中，双方各自需要配备一套证书，并拥有信任的签证机构的证书列表。使用私有签证机构颁发的数字证书时，用户通常需要手动将此私有签证机构的证书添加到信任的签证机构列表中。

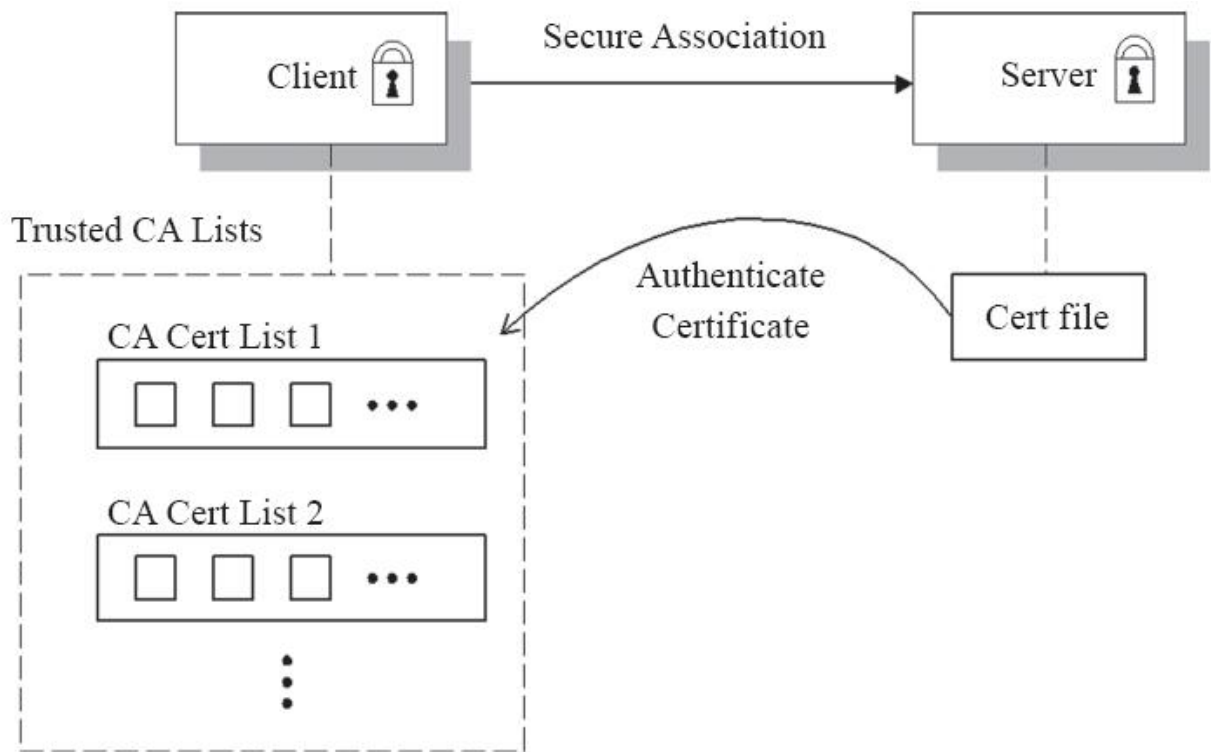


图10-2 SSL/TLS服务端认证（图片来源：Red Hat Inc.）

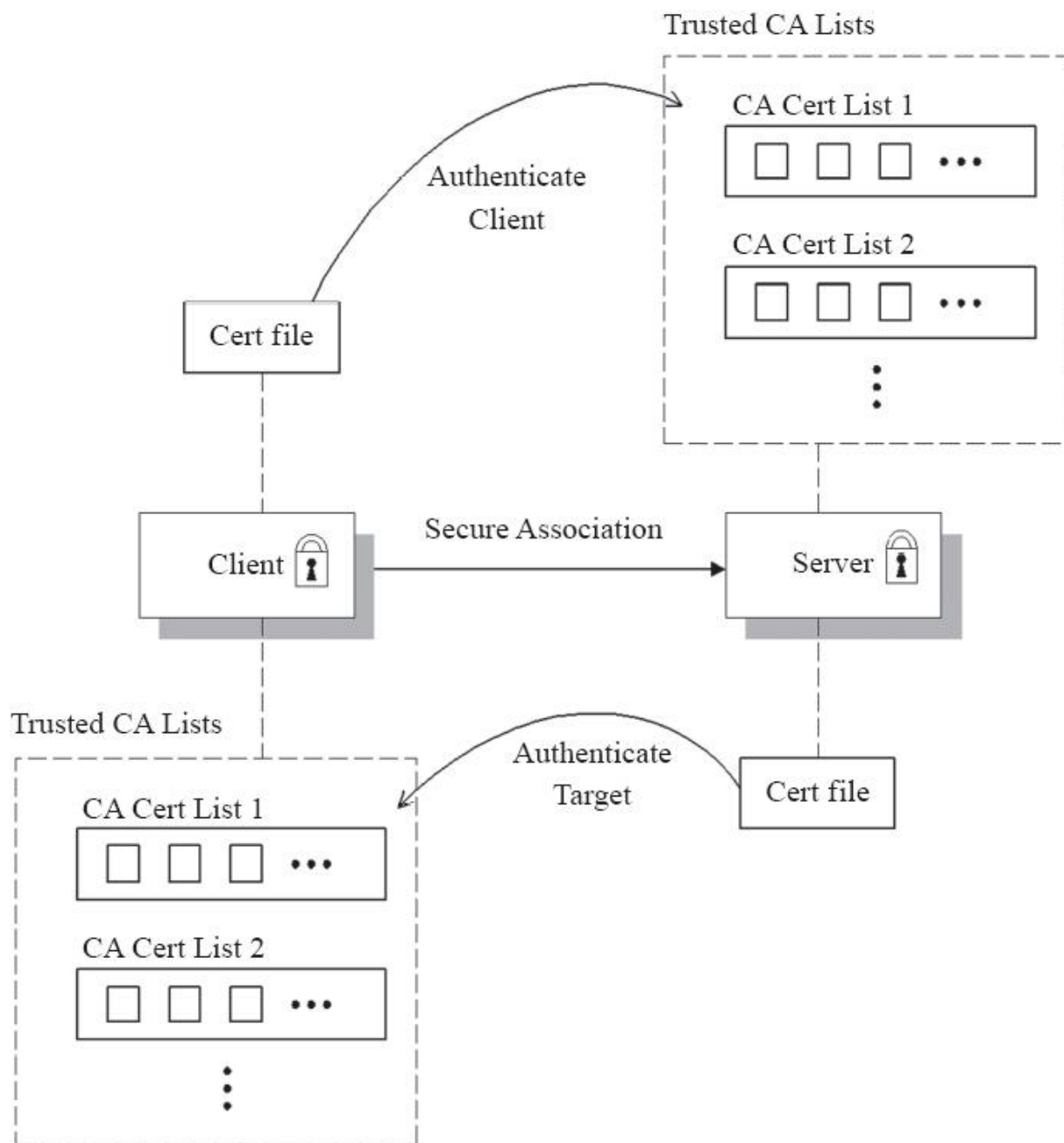


图10-3 SSL/TLS双向认证 (图片来源: Red Hat Inc.)

10.3.1 Kubernetes中的SSL/TLS认证

构建安全基础通信环境的Kubernetes集群时，需要用到TLS及数字证书的通信场景有很多种，如图10-4所示。API Server是整个Kubernetes集群的通信网关，controller-manager、scheduler、kubelet及kube-proxy等均需要经由API Server与etcd通信完成资源状态信息的获取及更新等。同样出于安全通信的目的，master的各组件（API Server、controller-manager和scheduler）需要基于SSL/TLS向外提供服务，而且与集群内部组件间进行通信时（主要是各节点上的kubelet和kube-proxy）还需要进行双向身份验证。

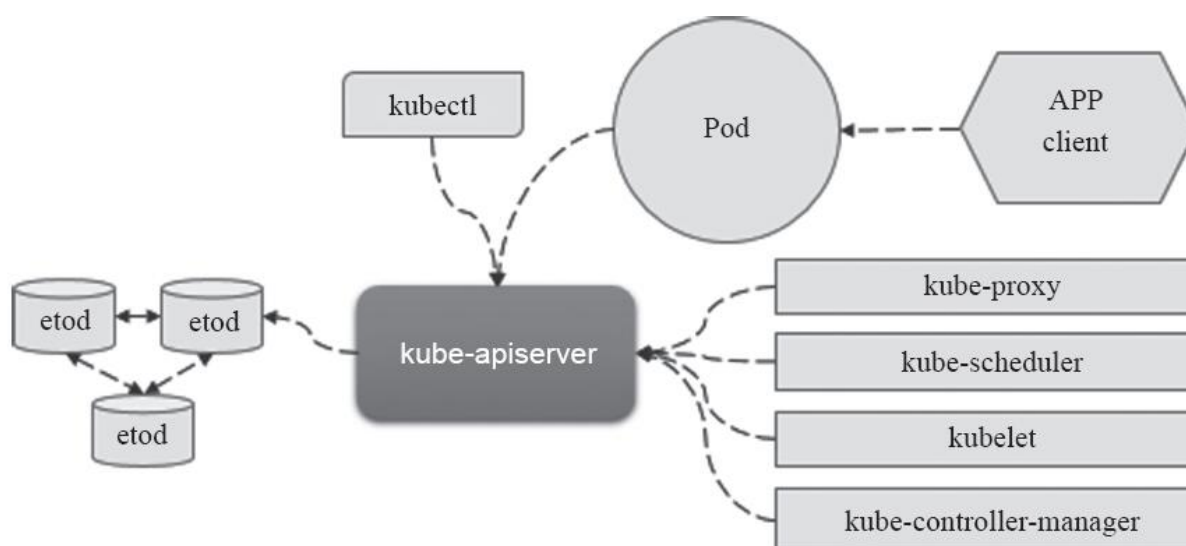


图10-4 Kubernetes的SSL/TLS通信

Kubernetes集群中各资源的状态信息，包括Secret对象中的敏感信息等均以明文方式存储于etcd中，因此，etcd集群内各节点间的通信，以及各节点与其客户端（主要是API server）之间的通信都应该以加密的方式进行，并需进行身份验证。

1) **etcd集群内对等节点通信**：etcd集群内各节点间的集群事务通信，默认监听于TCP的2380端口，基于SSL/TLS通信时需要peer类型的数字证书，可实现节点间的身份认证及通信安全，这些证书需要由一个专用CA进行管理。

2) **etcd服务器与客户端通信**: etcd的REST API服务, 默认监听于TCP的2379端口, 用于接收并响应客户端请求, 基于SSL/TLS通信, 支持服务端认证和双向认证, 而且要使用一个专用CA来管理此类证书; kube-apiserver就是etcd服务的主要客户端。

API Server与其客户端之间采用HTTPS通信可兼顾实现通信与认证的功能, 它们之间通信的证书可由同一个CA进行管理, 其客户端大体可以分为如下三类。

- 控制平面的kube-scheduler和kube-controller-manager。

- 工作节点组件kubelet和kube-proxy: 初次接入集群时, kubelet可自动生成私钥和证书签署请求, 并由Master为其自动进行证书签名和颁发, 这就是所谓的tls bootstrapping。

- kubelet及其他形式的客户端, 如Pod对象等。

集群上运行的应用 (Pod) 同其客户端的通信经由不可信的网络传输时也可能需要用到TLS/SSL协议, 如Nginx Pod与其客户端间的通信, 客户端来自于互联网时, 此处通常需要配置一个公信的服务端证书。

10.3.2 客户端配置文件kubeconfig

包括kubectl、kubelet和kube-controller-manager等在内的API Server的各类客户端都可以使用kubeconfig配置文件提供接入多个集群的相关配置信息，包括各API Server的URL及认证信息等，而且能够设置成不同的上下文环境，并在各环境之间快速切换，如图10-5所示。

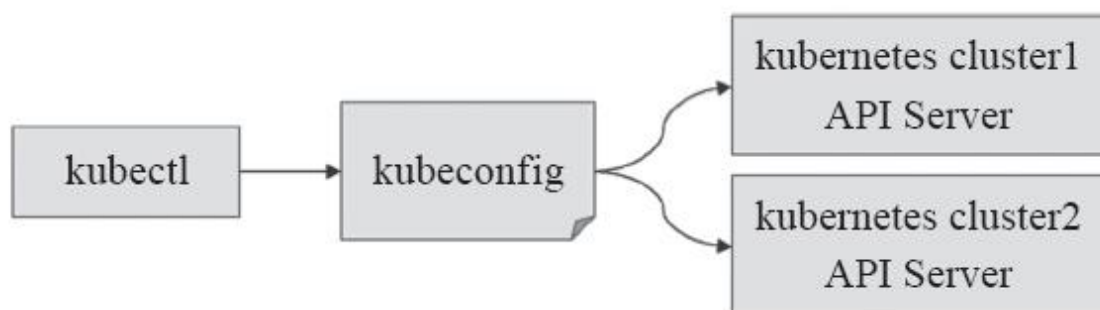


图10-5 kubectl和kubeconfig

使用kubeadm初始集群后生成的/etc/kubernetes/admin.conf文件即为kubeconfig格式的配置文件，其由kubeadm init命令自动生成，可由kubectl加载（默认路径为\$HOME/.kube/config）后用于接入服务器。“kubectl config view”命令能够显示当前正在使用的配置文件，下面的命令结果打印了文件中配置的集群列表、用户列表、上下文列表以及当前使用的上下文（current-context）等：

```
[root@master ~]# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://172.16.0.70:6443
    name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
    name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
```

```
client-certificate-data: REDACTED
client-key-data: REDACTED
```

事实上，任何类型的API Server客户端都可以使用kubeconfig进行配置，例如KubernetesNode之上的kubelet和kube-proxy也需要将其用到的认证信息保存于专用的kubeconfig文件中，并通过--kubeconfig选项进行加载。kubeconfig文件的定义中包含以下几项主要的配置，它们彼此之间的关系表示也由图10-6给出了。

- clusters: 集群列表，包含访问API Server的URL和所属集群的名称等。

- users: 用户列表，包含访问API Server时的用户名和认证信息。

- contexts: kubelet的可用上下文列表，由用户列表中的某特定用户名称和集群列表中的某特定集群名称组合而成。

- current-context: kubelet当前使用的上下文名称，即上下文列表中的某个特定项。

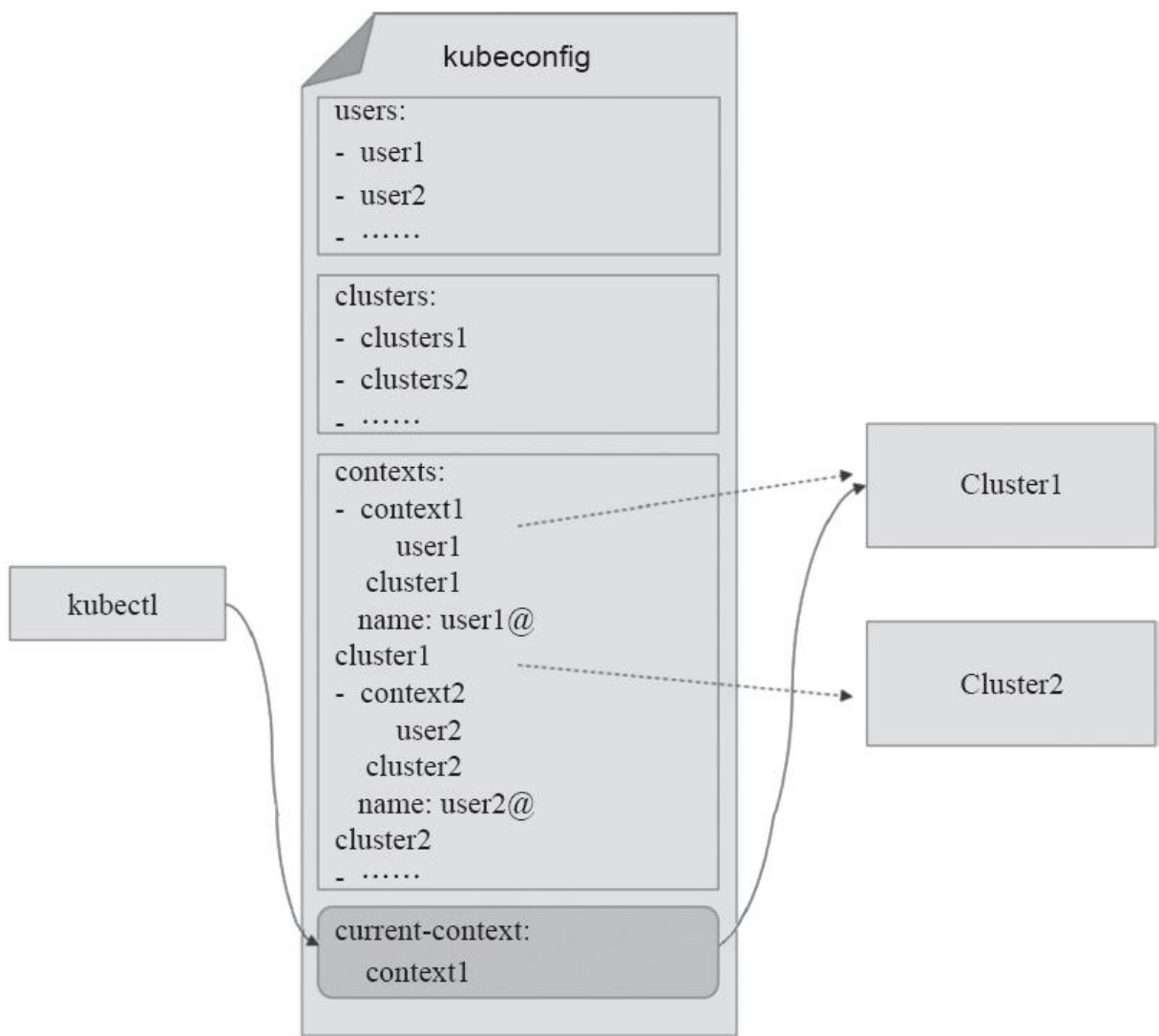


图10-6 kubeconfig文件格式示意图

用户也可以按需自定义相关的配置信息于**kubeconfig**配置文件中，以实现使用不同的用户账户接入集群等功能。**kubeconfig**是一个文本文件，虽然支持使用文本处理工具直接进行编辑，但这里建议用户使用“**kubectrl config**”命令进行设定，它能够自动进行语法检测等额外功能。**kubectrl config**命令的常用操作包含如下几项。

- kubectrl config view**: 打印**kubeconfig**文件内容。
- kubectrl config set-cluster**: 设置**kubeconfig**的**clusters**配置段。
- kubectrl config set-credentials**: 设置**kubeconfig**的**users**配置段。

·`kubectl config set-context`: 设置kubeconfig的contexts配置段。

·`kubectl config use-context`: 设置kubeconfig的current-context配置段。

使用kubeadm部署的Kubernetes集群默认提供了拥有集群管理权限的kubeconfig配置文件/etc/kubernetes/admin.conf，它可被复制到任何有着kubectl的主机上以用于管理整个集群。除此之外，管理员还可以创建其他基于SSL/TLS认证的自定义用户账号，以授予非管理员级的集群资源使用权限，其配置过程由两部分组成：一是为用户创建专用私钥及证书文件，二是将其配置于某kubeconfig文件中。下面给出其具体的实现过程，并借此创建将一个测试用户kube-user1用于后文中的授权测试。

第一步，为目标用户账号kube-user1创建私钥及证书文件，保存于/etc/kubernetes/pki目录中。



提示 本步骤中的操作需要在Master节点上以root用户的身份执行。

1) 生成私钥文件，注意其权限应该为600以阻止其他用户随意获取，这里在Master节点上以root用户进行操作，并将文件放置于/etc/kubernetes/pki专用目录中：

```
~]# cd /etc/kubernetes/pki
~]# (umask 077; openssl genrsa -out kube-user1.key 2048)
```

2) 创建证书签署请求，-subj选项中CN的值将被kubeconfig作为用户名使用，O的值将被识别为用户组：

```
~]# openssl req -new -key kube-user1.key -out kube-user1.csr -subj "/CN=kube-user1/
O=kubeusers"
```

3) 基于kubeadm安装Kubernetes集群时生成的CA签署证书，这里设置其有效时长为3650天：

```
~]# openssl x509 -req -in kube-user1.csr -CA ca.crt -CAkey ca.key \
-CACreateserial -out kube-user1.crt -days 3650
```

4) 验证证书信息（可选）：

```
~]# openssl x509 -in kube-user1.crt -text -noout
```

第二步，以默认的管理员kubernetes-admin@kubernetes为新建的kube-user1设定kube-config配置文件。配置结果将默认保存于当前系统用户的.kube/config文件中，当然也可以为kubectl使用--kubeconfig选项指定自定义的专用文件路径。

1) 配置集群信息，包括集群名称、API Server URL和CA证书，若集群信息已然存在，则可省略此步，另外，提供的新配置不能与现有配置中的集群名称相同，否则将覆盖它们：

```
~]$ kubectl config set-cluster kubernetes --embed-certs=true \
--certificate-authority=/etc/kubernetes/pki/ca.crt --server="https://172.16.
0.70:6443"
```

2) 配置客户端证书及密钥，用户名信息会通过命令从证书Subject的CN值中自动提取，例如前面创建csr时使用的“CN=kube-user1”，而组名则来自于“O=kubeusers”的定义：

```
~]$ kubectl config set-credentials kube-user1 --embed-certs=true \
--client-certificate=/etc/kubernetes/pki/kube-user1.crt \
--client-key=/etc/kubernetes/pki/kube-user1.key
```

3) 配置context，用来组合cluster和credentials，即访问的集群的上下文。如果为管理多个集群而设置了多个环境，则可以使用use-context来进行切换：

```
~]$ kubectl config set-context kube-user1@kubernetes --cluster=kubernetes --user=
kube-user1
```

4) 最后指定要使用的上下文，切换为以kube-user1访问集群：

```
~]$ kubectl config use-context kube-user1@kubernetes
```

5) 测试访问集群资源，不过在启用RBAC的集群上执行命令时，**kube-user1**并未获得集群资源的访问权限，因此会出现错误提示：

```
~]$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "kube-user1" cannot list
pods
in the namespace "default"
```

此时，若需切换至管理员账号，则可使用“**kubectl config use-context kubernetes-admin@kubernetes**”命令来完成。另外，临时使用某context时，不必设置current-context，只需要为kubectl命令使用--context选项指定目标context的名称即可，如“**kubectl--context= kube-user1@kubernetes get pods**”。

10.3.3 TLS bootstrapping机制

新的工作节点接入集群时需要事先配置好相关的证书和私钥等以进行安全通信，管理员既可以手动提供相关的配置，也可以选择由 **kubelet** 自行生成私钥和自签证书。集群略具规模后，第一种方式无疑会为管理员带来不小的负担，但其对于保障集群安全运行却又必不可少。第二种方式降低了管理员的工作量，却也损失了 **PKI** 本身所具有的诸多优势。综合上述两种方案，取长补短之后，**Kubernetes** 采用了一种新的方案：由 **kubelet** 自行生成私钥和证书签署请求，而后发送给集群上的证书签署进程（**CA**），由管理员验证请求后予以签署或直接控制器进程自动统一签署。这种方式即为 **Kubelet TLS Bootstrapping** 机制，它实现了前述第一种方式的功能，却基本上不会增加管理员工作量。

然而，启用了 **kubelet bootstrap** 功能之后，任何 **kubelet** 进程都可以向 **API Server** 发起签证请求并加入到集群中，包括那些来自于非计划或非授权主机的 **kubelet**，这必将增大管理签证操作时的识别工作量。此时，就需要配合使用某种认证机制对发出请求的 **kubelet** 加以认证，而后仅放行那些通过认证的签证请求。**API Server** 使用了一个基于认证令牌对“**system: bootstrappers**”组内的用户完成认证的认证器。**controller-manager** 会用到这个用户组配置默认的审批控制器（**approval controller**）适用的审批范围，它依赖于由管理员将此 **token** 正确绑定于 **RBAC** 策略，以限制其仅能用于签证操作相关的流程之中。



提示 **kubeadm** 启用了节点加入集群时的证书自动签署功能，因此加入过程在 **kubeadm join** 命令成功后即完成。

请求证书时，**kubelet** 要使用含有 **bootstrap token** 的 **kubeconfig** 文件向 **kube-apiserver** 发送请求，此 **kubeconfig** 在 **credentials** 中指定要调用的 **token** 文件名称必须是 **kubelet-bootstrap**。若指定的 **kubeocnfig** 配置文件不存在，则 **kubelet** 会转而使用 **bootstrap token** 从 **API Server** 中自动请求证书。证书请求审批通过后，**kubelet** 把接收到的证书和私钥的相关信息存储于 **--kubeconfig** 选项指定的文件中，而证书和私钥文件则存储于 **-cert-dir** 选项指定的目录下。

kube-controller-manager内部有一个用于证书颁发的控制循环，它是采用了类似于cfssl签发器格式的自动签发器，颁发的所有证书都仅有一年有效期限。签发依赖于CA提供加密组件支撑，此CA还必须被kube-apiserver的“--client-ca-file”选项指定的CA信任以用于认证。

Kubernetes 1.8之后的版本中使用的csrapproving审批控制器内置于kube-controller-manager中，并且默认为启用状态。此审批控制器使用SubjectAccessview API确认给定的用户是否有权限请求CSR（证书签署请求），而后再根据授权结果判定是否予以签署。不过，为了避免同其他审批器发生冲突，内建的审批器并不显式拒绝CSR，而只是忽略它们。

10.4 基于角色的访问控制：RBAC

RBAC（Role-Based Access Control，基于角色的访问控制）是一种新型、灵活且使用广泛的访问控制机制，它将权限授予“角色”（role）之上，这一点有别于传统访问控制机制中将权限直接赋予使用者的方式。

在RBAC中，用户（User）就是一个可以独立访问计算机系统中的数据或者用数据表示的其他资源的主体（Subject）。角色是指一个组织或任务中的工作或者位置，它代表了一种权利、资格和责任。许可（Permission）就是允许对一个或多个客体（Object）执行的操作。一个用户可经授权而拥有多个角色，一个角色可由多个用户构成；每个角色可拥有多种许可，每个许可也可授权给多个不同的角色。每个操作可施加于多个客体（受控对象），每个客体也可以接受多个操作。RBAC中的动作、主体及客体如图10-7所示。

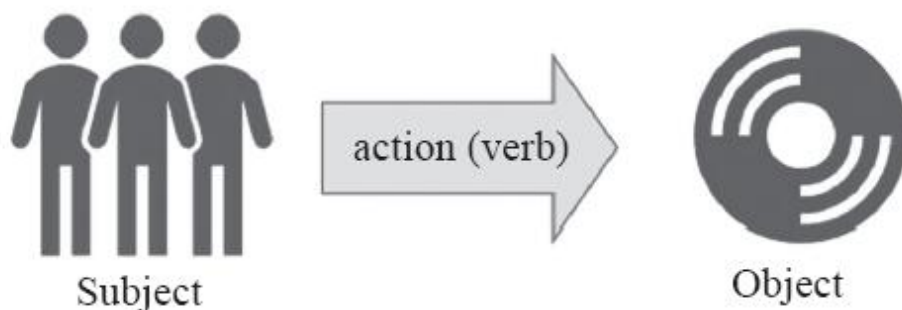


图10-7 RBAC中的主体、动作及客体

如其名字所示，RBAC的基于“角色”（role）这一核心组件实现了权限指派，具体实现中，它为账号赋予一到多个角色从而让其具有角色之上的权限，其中的账号可以是用户账号、用户组、服务账号及其相关的组等，而同时关联至多个角色的账号所拥有的权限是多个角色之上的权限集合。RBAC中的用户、角色及权限的关系如图10-8所示。

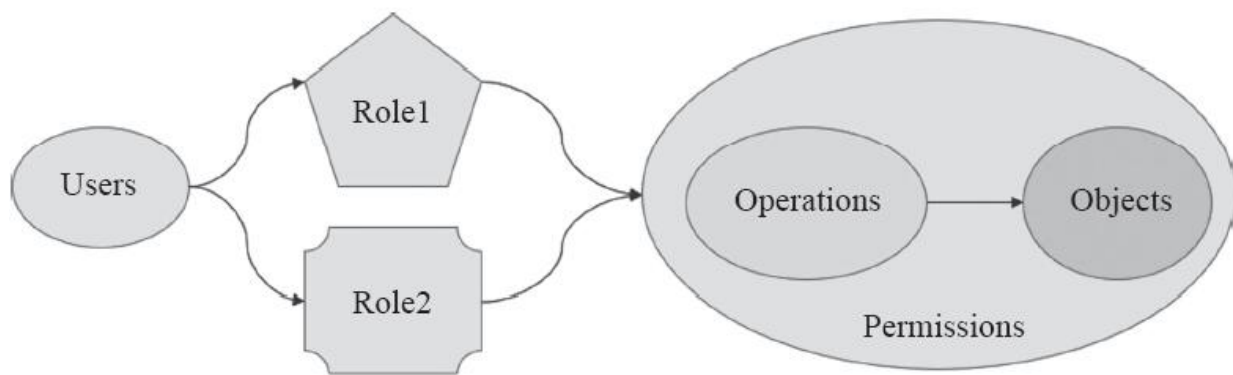


图10-8 RBAC中的用户、角色及权限

10.4.1 RBAC授权插件

RBAC是一种操作授权机制，用于界定“谁”（subject）能够或不能够“操作”（verb）哪个或哪类“对象”（object）。动作的发出者即“主体”，通常以“账号”为载体，它既可以是常规用户（User Account），也可以是服务账号（Service Account）。“操作”（verb）用于表明要执行的具体操作，包括创建、删除、修改和查看等，对应于kubectl来说，它通常由create、apply、delete、update、patch、edit和get等子命令来给出。而“客体”则是指操作施加于的目标实体，对Kubernetes API来说主要是指各类的资源对象以及非资源型URL。



提示 Kubernetes自1.5版起引入RBAC，1.6版本中将其升级为Beta级别，并成为kubeadm安装方式下的默认选项。而后，直到1.8版本，它才正式升级为stable级别。

相对于ABAC和Webhook等授权机制来说，RBAC具有如下优势。

- 1) 对集群中的资源和非资源型URL的权限实现了完整覆盖。
- 2) 整个RBAC完全由少数几个API对象实现，而且同其他API对象一样可以用kubectl或API调用进行操作。
- 3) 支持权限的运行时调整，无须重新启动API Server。

API Server是RESTful风格的API，各类客户端基于HTTP的请求报文（首部）发送身份认证信息并由认证插件完成身份验证，而后通过HTTP的请求方法指定对目标对象的操作请求并由授权插件进行授权检查，而操作的对象则是借助URL路径指定的REST资源。

图10-9对比给出了HTTP Verb和Kubernetes APIServer Verb的对应关系。

HTTP verb	request verb
POST	create
GET, HEAD	get (for individual resources), list (for collections)
PUT	update
PATCH	patch
DELETE	delete (for individual resources), deletecollection (for collections)

图10-9 HTTP Verb与API Server Verb

图10-10所描述的过程中，运行于API Server之上的授权插件RBAC负责确定某账号是否有权限对目标资源发出指定的操作，它们都属于“许可”（permission）类型的授权，不存在任何“拒绝”权限。

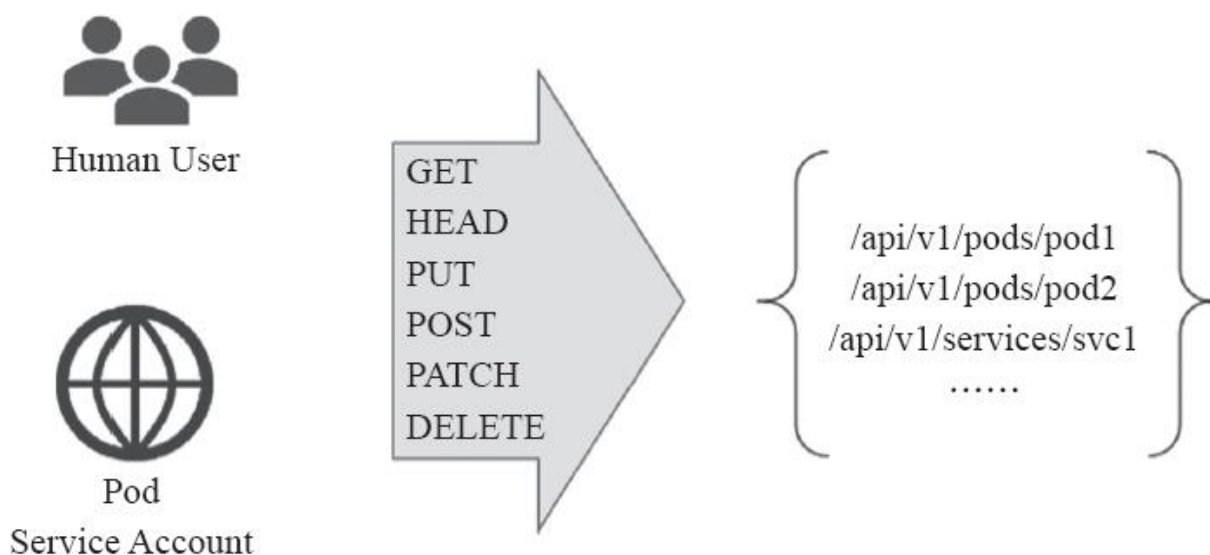


图10-10 Kubernetes RBAC简要模型

RBAC授权插件支持Role和ClusterRole两类角色，其中Role作用于名称空间级别，用于定义名称空间内的资源权限集合，而ClusterRole则用于组织集群级别的资源权限集合，它们都是标准的API资源类型。一般来说，ClusterRole的许可授权作用于整个集群，因此常用于控制Role无法生效的资源类型，这包括集群级别的资源（如Nodes）、非资源类型的端点（如/healthz）和作用于所有名称空间的资源（例如，跨名称空间获取任何资源的权限）。

对这两类角色进行赋权时，需要用到RoleBinding和ClusterRoleBinding这两种资源类型。RoleBinding用于将Role上的许可权限绑定到一个或一组用户之上，它隶属于且仅能作用于一个名称空间。绑定时，可以引用同一名称中的Role，也可以引用集群级别的ClusterRole。而ClusterRoleBinding则把ClusterRole中定义的许可权限绑定在一个或一组用户之上，它仅可以引用集群级别的ClusterRole。Role、RoleBinding、ClusterRole和ClusterRoleBinding的关系如图10-11所示。

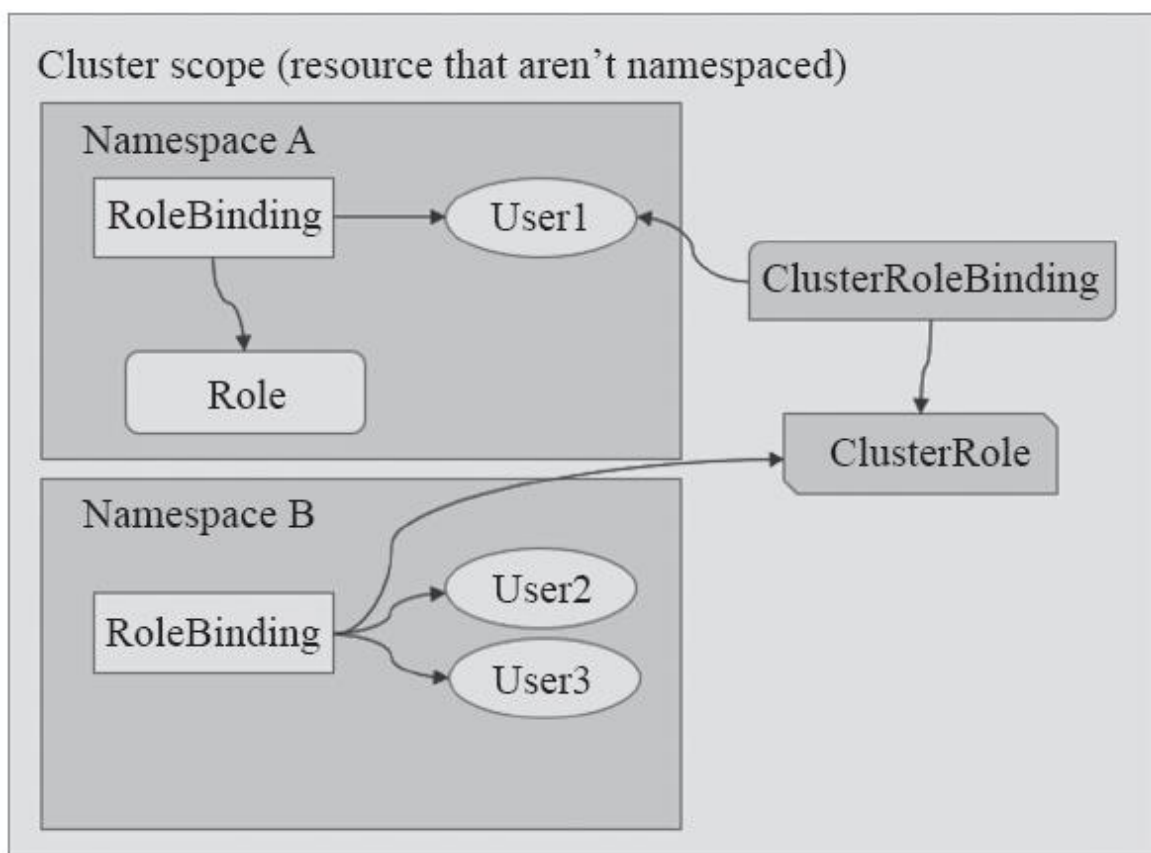


图10-11 Role、RoleBinding、ClusterRole和ClusterRoleBinding

一个名称空间中可以包含多个Role和RoleBinding对象，类似地，集群级别也可以同时存在多个ClusterRole和ClusterRoleBinding对象。而一个账户也可经由RoleBinding或ClusterRoleBinding关联至多个角色，从而具有多重许可授权。

10.4.2 Role和RoleBinding

Role仅是一组许可（**permission**）权限的集合，它描述了对哪些资源可执行何种操作，资源配置清单中使用**rules**字段嵌套授权规则。下面是一个定义在**testing**名称空间中的**Role**对象的配置清单示例，它设定了读取、列出及监视**Pod**和**Service**资源的许可权限：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: testing
  name: pods-reader
rules:
- apiGroups: ["" ] # "" 表示core API group
  resources: ["pods", "pods/log"]
  verbs: ["get", "list", "watch"]
```

类似上面的**Role**对象中的**rules**也称为**PolicyRule**，用于定义策略规则，不过它不包含规则应用的目标，其可以内嵌的字段包含如下几个。

1) **apiGroups**<[]string>: 包含了资源的**API**组的名称，支持列表格式指定的多个组，空串（""）表示核心组。

2) **resourceNames**<[]string>: 规则应用的目标资源名称列表，可选，缺省时意味着指定资源类型下的所有资源。

3) **resources**<[]string>: 规则应用的目标资源类型组成的列表，例如**pods**、**deployments**、**daemonsets**、**roles**等，**ResourceAll**表示所有资源。

4) **verbs**<[]string>: 可应用至此规则匹配到的所有资源类型的操作列表，可用选项有**get**、**list**、**create**、**update**、**patch**、**watch**、**proxy**、**redirect**、**delete**和**deletecollection**；此为必选字段。

5) **nonResourceURLs**<[]string>: 用于定义用户应该有权限访问的网址列表，它并非名称空间级别的资源，因此只能应用于**ClusterRole**

和ClusterRoleBinding，在Role中提供此字段的目的是仅为与ClusterRole在格式上兼容。

绝大多数资源均可通过其资源类型的名称引用，如“pods”或“services”等，这些名字与它们在API endpoint中的形式相同。不过，有些资源类型还支持子资源（subresource），例如Pod对象的/log，Node对象的/status等，它们的URL格式通常形如如下表示：

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

在RBAC角色定义中，如果要引用这种类型的子资源，则需要使用“resource/subresource”的格式，如上面示例规则中的“pods/log”。另外，还可以通过直接给定资源名称（resourceName）来引用特定的资源，此时支持使用的操作通常仅为get、delete、update和patch，也可使用这四个操作的子集实现更小范围的可用操作限制。

除了编写配置清单创建Role资源之外，还可以直接使用“kubectl create role”命令进行快速创建，例如，下面的命令在testing名称空间中创建了一个名为services-admin的角色：

```
~]$ kubectl create role services-admin --verb="*" --resource="services, services/*" -n testing
```

将清单中的Role资源也创建于集群上之后，testing名称空间中就有了pods-reader和services-admin两个角色，但角色本身并不能作为动作的执行主体，它们需要“绑定”（Role-Binding）到主体（如user、group或服务account）之上才能发生作用。

RoleBinding用于将Role中定义的权限赋予一个或一组用户，它由一组主体，以及一个要引用来赋予这组主体的Role或ClusterRole组成。需要注意的是，RoleBinding仅能够引用同一名称空间中的Role对象完成授权，例如，下面配置清单中的RoleBinding于testing名称空间中将pods-reader角色赋给了用户kube-user1，从而使得kube-user1拥有了此角色之上的所有许可授权：

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
  name: resources-reader
  namespace: testing
subjects:
- kind: User
  name: kube-user1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pods-reader
  apiGroup: rbac.authorization.k8s.io
```

将此RoleBinding资源应用到集群中， kube-user1用户便有了读取testing名称空间中pods资源的权限。将kubectl的配置上下文切换至kube-user1用户，分别对pods和services资源发起访问请求测试，由下面的命令结果可以看出，它能够请求读取pods资源，但对其他任何资源的任何操作请求都将被拒绝：

```
~]$ kubectl config use-context kube-user1@kubernetes
Switched to context "kube-user1@kubernetes".
~]$ kubectl get pods -n testing
No resources found.
~]$ kubectl get services -n testing
Error from server (Forbidden): services is forbidden: User "kube-user1" cannot
list services in the namespace "testing"
```

RoleBinding资源也能够直接在命令行中创建。例如，将kubectl的上下文切换为kubernetes-admin用户，将前面创建的services-admin角色绑定于kube-user1之上，可以使用如下命令进行：

```
~]$ kubectl config use-context kubernetes-admin@kubernetes
~]$ kubectl create rolebinding admin-services --role=services-admin --user=kube-
user1 -n testing
rolebinding.rbac.authorization.k8s.io "admin-services" created
```

再次切换到kube-user1用户，进行services资源的访问测试，由下面的命令结果可知，它能够访问services资源，而不再是拒绝权限：

```
~]$ kubectl config use-context kube-user1@kubernetes
~]$ kubectl get services -n testing
No resources found.
```

事实上，通过admin-services这个RoleBinding绑定至services-admin角色之后，kube-user1用户拥有管理testing名称空间中的Service资源的所有权限。另外需要注意的是，虽然RoleBinding不能跨名称空间引用Role资源，但主体中的用户账号、用户组和服务账号却不受名称空间的限制，因此，管理员可为一个主体通过不同的RoleBinding资源绑定多个名称空间中的角色。

RoleBinding的配置中主要包含两个嵌套的字段subjects和roleRef，其中，subjects的值是一个对象列表，用于给出要绑定的主体，而roleRef的值是单个对象，用于指定要绑定的Role或ClusterRole资源。subjects字段的可嵌套字段具体如下。

- apiGroup<string>：要引用的主体所属的API群组，对于ServiceAccount类的主体来说默认为""，而User和Group类主体的默认值为"rbac.authorization.k8s.io"。

- kind<string>：要引用的资源对象（主体）所属的类别，可用值为"User""Group"和"ServiceAccount"三个，必选字段。

- name<string>：引用的主体的名称，必选字段。

- namespace<string>：引用的主体所属的名称空间，对于非名称空间类型的主体，如"User"和"Group"，其值必须为空，否则授权插件将返回错误信息。

roleRef的可嵌套字段具体如下。

- apiGroup<string>：引用的资源（Role或ClusterRole）所属的API群组，必选字段。

- kind<string>：引用的资源所属的类别，可用值为Role或ClusterRole，必选字段。

- name<string>：引用的资源的名称。

Role和RoleBinding是名称空间级别的资源，它们仅能用于完成单个名称空间内的访问控制，此时若要赋予某主体多个名称空间中的访问权限，就不得不逐个名称空间地进行。另外还有一些资源其本身并

不属于名称空间（如PersistentVolume、Namespace和Node等），甚至还有一些非资源型的URL路径（如/healthz等），对此类资源的管理显然无法在名称空间级别完成，此时就需要用到另外两个集群级别的资源类型ClusterRole和ClusterRoleBinding。

10.4.3 ClusterRole和ClusterRoleBinding

集群级别的角色资源ClusterRole资源除了能够管理与Role资源一样的许可权限之外，还可以用于集群级组件的授权，配置方式及其在rules字段中可内嵌的字段也与Role资源类似。下面的配置清单示例中定义了ClusterRole资源nodes-reader，它拥有访问集群的节点信息的权限：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nodes-reader
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get", "watch", "list"]
```

ClusterRole是集群级别的资源，它不属于名称空间，故在此处其配置不应该使用metadata.namespace字段，这也可以通过获取nodes-reader的状态信息来进行验证。kube-user1用户此前绑定的pods-reader和services-admin角色属于名称空间，它们无法给予此用户访问集群级别nodes资源的权限。

RoleBinding也能够将主体绑定至ClusterRole资源之上，但仅能赋予用户访问Role-Binding资源本身所在的名称空间之内可由ClusterRole赋予的权限，例如，在ClusterRole具有访问所有名称空间的ConfigMap资源权限时，通过testing名称空间的RoleBinding将其绑定至kube-user1用户，则kube-user1用户就具有了访问testing名称空间中的ConfigMap资源的权限，但不能访问其他名称空间中的ConfigMap资源。不过，若借助ClusterRoleBinding进行绑定，则kube-user1就具有了所有相关名称空间中的资源的访问权限，如图10-12所示。

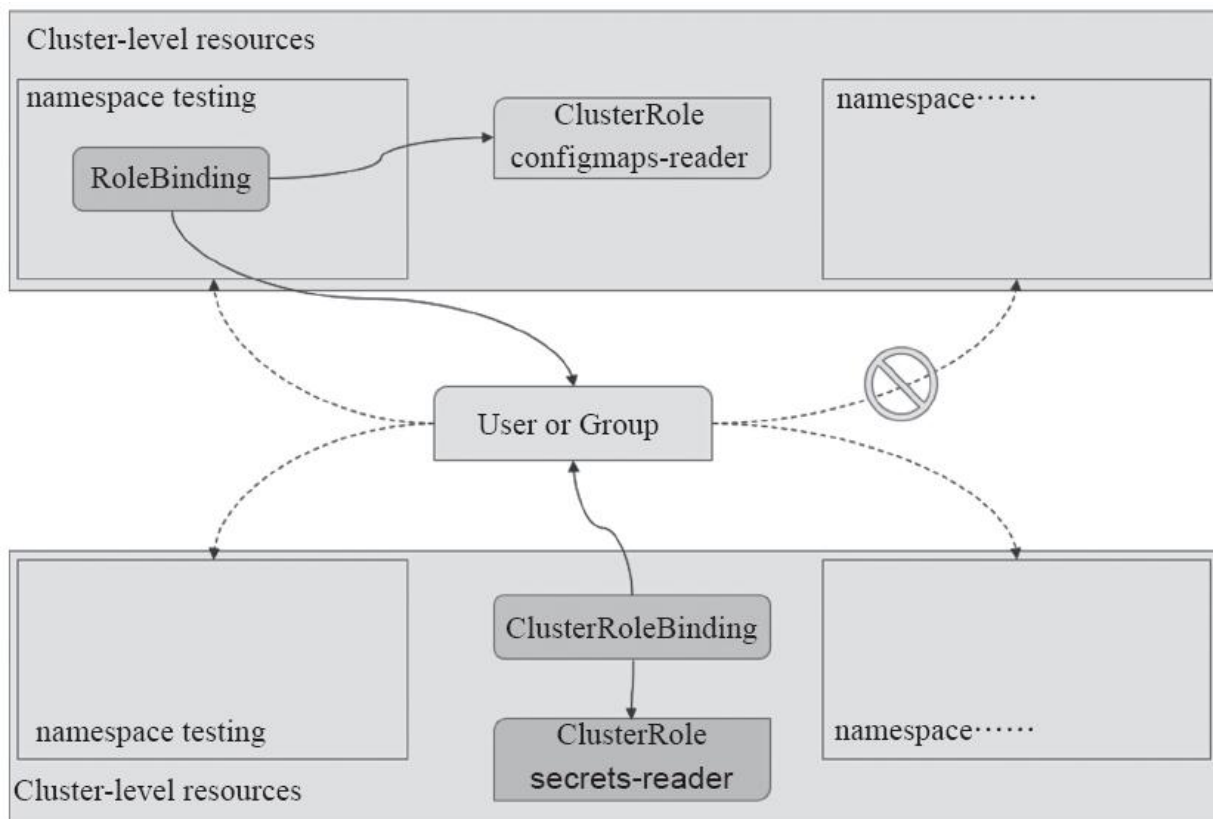


图10-12 RoleBinding与ClusterRoleBinding

因此，一种常见的做法是集群管理员在集群范围内预先定义好一组访问名称空间级别资源权限的ClusterRole资源，而后在多个名称空间中多次通过RoleBinding引用它们，从而让用户分别具有不同名称空间上的资源的相应访问权限，完成名称空间级别权限的快速授予。当然，如果通过ClusterRoleBinding引用这类的ClusterRole，则相应的用户即拥有了在所有相关名称空间上的权限。

集群级别的资源nodes、persistentvolumes等资源，以及非资源型的URL不属于名称空间级别，故此通过RoleBinding绑定至用户时无法完成访问授权。事实上，所有的非名称空间级别的资源都无法通过RoleBinding绑定至用户并赋予用户相关的权限，这些是属于ClusterRoleBinding的功能。

另外，除了名称空间及集群级别的资源之外，Kubernetes还有着/api、/apis、/healthz、/swaggerapi和/version等非资源型URL，对这些URL的访问也必须事先获得相关的权限。同集群级别的资源一样，它们也只能定义在ClusterRole中，且需要基于ClusterRoleBinding进行授

权。不过，对此类资源的读取权限已经由系统默认的名称同为**system: discovery**的**ClusterRole**和**ClusterRoleBinding**两个资源自动设定。下面的命令显示了**system: discovery ClusterRole**的相关信息：

```
~]$ kubectl get clusterrole system:discovery -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:discovery
...
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /swagger-2.0.0.pb-v1
  - /swagger.json
  - /swaggerapi
  - /swaggerapi/*
  - /version
verbs:
- get
```

nonResourceURLs字段给出了相关的URL列表（不同版本的Kubernetes系统上的显示可能略有区别），允许的访问权限仅有“**get**”一个。引用了此资源的**ClusterRoleBinding**的相关信息如下面的命令及其结果所示：

```
~]$ kubectl get clusterrolebindings system:discovery -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:discovery
...
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:unauthenticated
```

由命令的输出结果可知，它绑定了**system: authenticated**和**system: unauthencated**两个组，这囊括了所有的用户账号，因此所有用

户默认均有权限请求读取这些资源，任何发往API Server的此类端点读取请求都会得到响应。定义其他类型的访问权限，其方法可参照 **system: discovery** 这个ClusterRole资源进行。例如，下面的ClusterRole资源示例定义了对非资源型URL路径/healthz的读写访问权限：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: healthz-admin
rules:
- nonResourceURLs:
  - /healthz
  verbs:
  - get
  - create
```

另外，非资源型URL的授权规则与资源权限的授权规则可定义在同一个ClusterRole中，它们同属于rules字段中的对象。

10.4.4 聚合型ClusterRole

Kubernetes自1.9版本开始支持在rules字段中嵌套aggregationRule字段来整合其他的ClusterRole对象的规则，这种类型的ClusterRole的权限受控于控制器，它们由所有被标签选择器匹配到的用于聚合的ClusterRole的授权规则合并生成。下面是一个示例，它定义了一个标签选择器用于挑选匹配的ClusterRole：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: []
```

任何能够被示例中的资源的标签选择器匹配到的ClusterRole的相关规则都将一同合并为它的授权规则，后续新增的ClusterRole资源亦是如此。因此，聚合型ClusterRole的规则会随着标签选择器的匹配结果而动态变化。下面即是一个能匹配到此聚合型ClusterRole的另一个ClusterRole的示例：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# These rules will be added to the "monitoring" role.
rules:
- apiGroups: [""]
  Resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

创建这两个资源，而后查看聚合型ClusterRole资源monitoring的相关权限信息，由下面的命令结果可知，它所拥有的权限包含了ClusterRole资源monitoring-endpoints的所有授权：

```
~]$ kubectl create -f monitoring.yaml -f monitoring-endpoints.yaml
~]$ kubectl get clusterrole monitoring -o yaml
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: monitoring
...
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
```

事实上，Kubernetes系统上面向用户的内建ClusterRole `admin`和`edit`也是聚合型Cluster-Role，因为这可以使得默认角色中包含自定义资源的相关规则，例如，由CustomResource-Definitions或Aggregated API服务器提供的规则等。

10.4.5 面向用户的内建ClusterRole

API Server内建了一组默认的ClusterRole和ClusterRoleBinding以预留系统使用，其中大多数都以“system: ”为前缀。另外还有一些非以“system: ”为前缀的默认的Role资源，它们是为面向用户的需求而设计的，包括超级用户角色（cluster-admin）、用于授权集群级别权限的ClusterRoleBinding（cluster-status）以及授予特定名称空间级别权限的RoleBinding（admin、edit和view），如图10-13所示。掌握这些默认的内建ClusterRole对后期按需创建用户并快速配置权限至关重要。

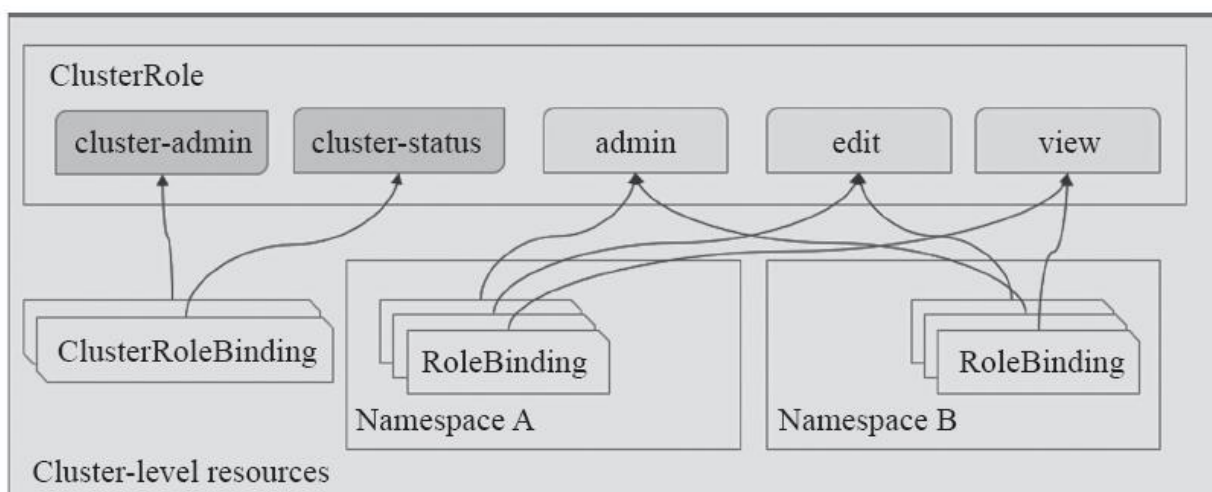


图10-13 内建的面向用户的ClusterRole

内建的ClusterRole资源cluster-admin拥有管理集群所有资源的权限，它基于同名的ClusterRoleBinding资源绑定到了“system: masters”组上，这意味着所有隶属于此组的用户都将具有集群的超级管理权限。kubeadm安装设定集群时自动创建的配置文件/etc/kubernetes/admin.conf中定义的用户kubernetes-admin使用证书文件/etc/kubernetes/pki/apiserver-kubelet-client.crt向API Server进行验证。而此证书的Subject信息为“/O=system: masters/CN=kubernetes-admin”，进行认证时，CN的值可作为用户名使用，而O的值将作为用户所属组名使用，因此，kubernetes-admin具有集群管理权限。

于是，为Kubernetes集群额外自定义超级管理员的方法至少具有两种：一种是创建用户证书，其Subject中O的值为“system: masters”，另一种是创建ClusterRoleBinding将用户绑定至cluster-admin之上。其具体

的实现方法均可参照10.4.4节和本节中的内容略加变通而实现，cluster-status的使用方法与此类同，有所区别的仅在于权限本身。

另外，在多租户、多项目或多环境等使用场景中，用户通常应该获取名称空间级别绝大多数资源的管理、只读或编辑权限，这类权限的快速授予可通过在指定的名称空间中创建RoleBinding资源引用内建的ClusterRole资源admin、view或edit来进行。例如，在名称空间dev中创建一个RoleBinding资源，它将引用admin，并绑定至kube-user1，使得此用户具有管理dev名称空间中除了名称空间本身及资源配额之外的所有资源的权限：

```
~]$ kubectl create rolebinding dev-admin --clusterrole=admin --user=kube-user1 -n dev
```

而后切换至kube-user1用户在dev中创建一个资源，并查看相关的信息进行访问测试，如下面的命令所示：

```
~]$ kubectl create deployment myapp-deploy --image=ikubernetes/myapp:v1 -n dev
deployment.extensions "myapp-deploy" created
~]$ kubectl get all -n dev
.....
```

如果需要授予的是编辑或只读权限，则仅需要将创建RoleBinding时引用的ClusterRole相应地修改为edit或view即可。表10-1总结了典型的面向用户的内建ClusterRole及其功用。

表10-1 面向用户的内建ClusterRole资源

默认的 ClusterRole	默认的 ClusterRoleBinding	说明
cluster-admin	system:masters 组	授予超级管理员在任何对象上执行任何操作的权限
admin	None	以 RoleBinding 机制访问指定名称空间的所有资源，包括名称空间的 Role 和 RoleBinding，但不包括资源配置和名称空间本身
edit	None	允许读写访问一个名称空间内的绝大多数资源，但不允许查看或修改 Role 或 RoleBinding
view	None	允许读取一个名称空间内的绝大多数资源，但不允许查看 Role 或 RoleBinding，以及 secret 资源

10.4.6 其他的内建ClusterRole和ClusterRoleBinding

API Server会创建一组默认的ClusterRole和ClusterRoleBinding，大多数都以“system: ”为前缀，被系统基础架构所使用，修改这些资源将会导致集群功能不正常。例如，如果修改了为kubelet赋权的“system: node”这一ClusterRole，则将会导致kubelet无法工作。所有默认的ClusterRole和ClusterRoleBinding都打了标签“kubernetes.io/bootstrapping=rbac-defaults”。

每次启动时，API Server都会自动为默认的ClusterRole重新赋予缺失的权限，以及为默认的ClusterRoleBinding绑定缺失的Subject。这种机制给了集群从意外修改中自动恢复的能力，以及升级版本后，自动将ClusterRole和ClusterRoleBinding升级到满足新版本需求的能力。



提示 必要时，在默认的ClusterRole或ClusterRoleBinding上设置annotation中“rbac.authorization.kubernetes.io/autoupdate”属性的值为“false”即可禁止此种自动恢复功能。

Kubernetes内建的其他ClusterRole和ClusterRoleBinding还有很多，它们绝大多数都是用于系统本身的诸多组件结合RBAC获得最小化但完整的资源访问授权。它们大体可分为专用于核心的组件、附加的组件、资源发现及controller-manager运行核心控制循环（control loop）等几个类型，如system: kube-scheduler、system: kube-controller-manager、system: node、system: node-proxier和system: kube-dns等，大多数都可以做到见名知义，这里不再逐一给出说明。

10.5 Kubernetes Dashboard

Dashboard是Kubernetes的Web GUI，可用于在Kubernetes集群上部署容器化应用、应用排障、管理集群本身及其附加的资源等。它常被管理员用于集群及应用速览、创建或修改单个资源（如Deployments、Jobs和DaemonSets等），以及扩展Deployment、启动滚动更新、重启Pod或使用部署向导部署一个新应用等。



注意 Dashboard依赖于Heapster或Metrics Server完成指标数据的采集和可视化。

Dashboard的认证和授权均可由Kubernetes集群实现，它自身仅是一个代理，所有的相关操作都将发给API Server进行，而非由Dashboard自行完成。目前它支持使用的认证方式有承载令牌（bearer token）和kubeconfig两种，在访问之前需要准备好相应的认证凭证。

10.5.1 部署HTTPS通信的Dashboard

Dashboard 1.7（不含）之前的版本在部署时直接赋予了管理权限，这种方式可能存在安全风险，因此，1.7及之后的版本默认在部署时仅定义了运行Dashboard所需要的最小权限，仅能够在Master主机上通过“**kubectl proxy**”命令创建代理后于本机进行访问，它默认禁止了来自于其他任何主机的访问请求。若要绕过“**kubectl proxy**”直接访问Dashboard，则需要在部署时提供证书以便与客户端进行通信时建立一个安全的HTTPS连接。证书既可由公信CA颁发，也可自行使用openssl或cfssl一类的工具来创建。

部署Dashboard时会从Secrets对象中加载所需要的私钥和证书文件，需要事先准备好相关的私钥、证书和Secrets对象。需要注意的是，下面的命令需要以管理员的身份在Master节点上运行，否则将无法访问到ca.key文件：

```
~]# (umask 077; openssl genrsa -out dashboard.key 2048)
~]# openssl req -new -key dashboard.key -out dashboard.csr -subj
"/O=iLinux/CN=dashboard"
~]# openssl x509 -req -in dashboard.csr -CA /etc/kubernetes/pki/ca.crt \
    -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out dashboard.crt -days
3650
```

接下来，基于生成的私钥和证书文件创建名为kubernetes-dashboard-certs的Opaque类型的Secret对象，其键名分别为dashboard.key和dashboard.crt：

```
~]$ kubectl create secret generic kubernetes-dashboard-certs \
    -n kube-system --from-file=dashboard.crt=./dashboard.crt \
    --from-file=dashboard.key=./dashboard.key -n kube-system
```

Secret对象准备完成后即可部署Dashboard。若无其他自定义部署的需要，可直接基于在线资源配置清单。不过，资源清单中也创建了Secrets对象，因此，它可能会发出警告信息。另外，默认创建的Service对象类型为ClusterIP，它仅能在Pod客户端中访问，若需在集群外通过浏览器访问Dashboard，则需要修改其类型为NodePort再进行创

建，或者在创建后通过修改命令进行设定。这里采取直接在线创建，并修改其Service对象kubernetes-dashboard的类型的方式：

```
~]$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/
src/deploy/recommended/kubernetes-dashboard.yaml
~]$ kubectl patch svc kubernetes-dashboard -p '{"spec":{"type":"NodePort"}}' -n
kube-system
```

确认其使用的NodePort之后便可在集群外通过浏览器进行访问：

```
~]$ kubectl get svc kubernetes-dashboard -n kube-system
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes-dashboard NodePort    10.110.145.88 <none>         443:31999/TCP    2h
```

图10-14即为其默认显示的登录页面，支持的认证方式为kubernetes-dashboard和token（令牌）两种。



图10-14 Dashboard认证界面

Dashboard是运行于Pod对象中的应用，其连接API Server的账户应为ServiceAccount类型，因此，用户在登录界面提供的用户账号须得是此类账户，而且访问权限也取决于kubernetes-dashboard或token认证时的ServiceAccount用户的权限。ServiceAccount的集群资源访问权限取决于它绑定的角色或集群角色，例如，为登录的用户授权集群级别的管理权限时可直接绑定内建的集群角色cluster-admin，授权名称空间级别的

管理权限时，可直接绑定内建的集群角色**admin**。当然，也可以直接自定义**RBAC**的角色或集群角色，并进行绑定授权，例如集群级别的只读权限或名称空间的只读权限。

10.5.2 配置token认证

集群级别的管理操作依赖于集群管理员权限，例如，管理持久存储卷和名称空间等资源，内建的cluster-admin集群角色拥有相关的全部权限，创建ServiceAccount并将其绑定其上即可完成集群管理员授权。而用户通过相应的ServiceAccount的token信息完成Dashboard认证也就能扮演起Dashboard接口上的集群管理员角色。例如，下面创建一个名为dashboard-admin的ServiceAccount，并完成集群角色绑定：

```
~]$ kubectl create serviceaccount dashboard-admin -n kube-system
~]$ kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin
\
--serviceaccount=kube-system:dashboard-admin
```

创建ServiceAccount对象时，它会自动为用户生成用于认证的token信息，这一点可以从与其相关的Secrets对象上获取：

```
~]$ ADMIN_SECRET=$(kubectl -n kube-system get secret | awk '/^dashboard-
admin/{print $1}')
~]$ kubectl describe secrets $ADMIN_SECRET -n kube-system kube-system
Name:          dashboard-admin-token-jfcd6
Namespace:     kube-system
.....
token:         eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3N1cnZ
pY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrd
WJlLXN5c3RlbSIsImt1YmVyb.....
```

对上面获取到的token在登录界面选择令牌认证方式，并键入token令牌即要登录的Dashboard，其效果如图10-15所示。

上面认证时用到的令牌将采用base64的编码格式，且登录时需要将原内容贴入文本框，存储及操作都有其不便之处，建议用户将它存入kubeconfig配置文件中，通过kubeconfig的认证方式进行登录。

10.5.3 配置kubeconfig认证

kubeconfig是认证信息承载工具，它能够存入私钥和证书，或者认证令牌等作为用户的认证配置文件。为了说明如何配置一个仅具有特定名称空间管理权限的登录账号，这里再次创建一个新的**ServiceAccount**用于管理默认的**default**名称空间，并将之绑定于**admin**集群角色，其操作过程与10.5.2节的方式相似：

```
~]$ kubectl create serviceaccount def-ns-admin -n default
~]$ kubectl create rolebinding def-ns-admin --clusterrole=admin \
  --serviceaccount=default:def-ns-admin
```



图10-15 Dashboard主面板

下面分步骤说明如何创建所需的**kubeconfig**文件。第一步，初始化集群信息，提供**API Server**的URL，以及验证**API Server**证书所用到的**CA证书**等。

```
~]$ kubectl config set-cluster kubernetes --embed-certs=true --server="https://
172.16.0.70:6443" \
  --certificate-authority=/etc/kubernetes/pki/ca.crt --kubeconfig=./def-ns-
```

```
admin.  
    kubeconfig
```

第二步，获取def-ns-admin的token，并将其作为认证信息。由于直接得到的token为base64编码格式，因此，下面使用了“base64-d”命令将其解码还原：

```
~]$ DEFNS_ADMIN_SECRET=$(kubectl -n default get secret | awk '/^def-ns-admin  
/{print $1}')
```

```
~]$ DEFNS_ADMIN_TOKEN=$(kubectl -n default get secret ${DEF_NS_ADMIN_SECRET} \  
-o jsonpath={.data.token}|base64 -d)  
~]$ kubectl config set-credentials def-ns-admin --token=${DEFNS_ADMIN_TOKEN} \  
--kubeconfig=./defns-admin.kubeconfig
```

第三步，设置context列表，定义一个名为def-ns-admin的context：

```
~]$ kubectl config set-context def-ns-admin --cluster=kubernetes \  
--user=defns-admin --kubeconfig=./defns-admin.kubeconfig
```

最后指定要使用的context为前面定义的名为def-ns-admin的context：

```
~]$ kubectl config use-context def-ns-admin --kubeconfig=./defns-admin.kubeconfig
```

到此为止，一个用于Dashboard登录认证的default名称空间的管理员账号配置文件已经设置完成，将文件复制到远程客户端上即可用于登录认证。

配置token认证和kubeconfig认证时创建ServiceAccount和完成集群角色的绑定以及角色绑定的过程也可以使用资源配置清单文件来实现，相关的文件可在本章的代码清单目录中获得，分别为dashboard-admin.yaml和def-ns-admin.yaml，可用它们取代前面相关的创建命令。关于Dashboard的使用这里就不再多做介绍了，读者根据界面提示信息很快就能掌握共用法。

10.6 准入控制器与应用示例

在经由认证插件和授权插件分别完成身份认证和权限检查之后，准入控制器将拦截那些创建、更新和删除相关的操作请求以强制实现控制器中定义的功能，包括执行对象的语义验证、设置缺失字段的默认值、限制所有容器使用的镜像文件必须来自某个特定的Registry、检查Pod对象的资源需求是否超出了指定的限制范围等。

但是准入控制器的相关代码必须要由管理员编译进kube-apiserver中才能使用，实现方式缺乏灵活性。于是，Kubernetes自1.7版本引入了Initializers和External Admission Webhooks来尝试突破此限制，而且自1.9版本起，External Admission Webhooks又被分为Mutating-AdmissionWebhook和ValidatingAdmissionWebhook两种类型，分别用于在API中执行对象配置的“变异”和“验证”操作。在具体的代码实现上，一个准入控制器可以是“验证”型、“变异”型或兼具此两项功能。变异控制器可以修改他们许可的对象，而验证控制器则一般不会。

在具体运行时，准入控制可分为两个阶段，第一个阶段串行运行各变异型控制器，第二个阶段串行运行各验证型控制器，在此过程中，一旦任一阶段中的任何控制器拒绝请求，则立即拒绝整个请求，并向用户返回错误。

10.6.1 LimitRange资源与LimitRanger准入控制器

虽然用户可以为容器指定资源需求及资源限制，但未予指定资源限制属性的容器应用很有可能因故吞掉所在工作节点上的所有可用计算资源，因此妥当的做法是使用**LimitRange**资源在每个名称空间中为每个容器指定最小及最大计算资源用量，甚至是设置默认的计算资源需求和计算资源限制。在名称空间上定义了**LimitRange**对象之后，客户端提交创建或修改的资源对象将受到**LimitRanger**控制器的检查，任何违反**LimitRange**对象定义的资源最大用量的请求将被直接拒绝。

LimitRange资源支持限制容器、Pod和PersistentVolumeClaim三种资源对象的系统资源用量，其中Pod和容器主要用于定义可用的CPU和内存资源范围，而**PersistentVolume-Claim**则主要定义存储空间的限制范围。下面的配置清单以容器的CPU资源为例，**default**用于定义默认的资源限制，**defaultRequest**定义默认的资源需求，**min**定义最小的资源用量，而最大的资源用量既可以使用**max**给出固定值，也可以使用**maxLimitRequestRatio**设定为最小用量的指定倍数：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1000m
    defaultRequest:
      cpu: 1000m
    min:
      cpu: 500m
    max:
      cpu: 2000m
    maxLimitRequestRatio:
      cpu: 4
  type: Container
```

将配置清单中的资源创建于集群上的**default**名称空间中，而后即可使用**describe**命令查看相关资源的生效结果。创建不同的Pod对象对默认值、最小资源限制及最大资源限制分别进行测试以验证其限制机制。

首先，创建一个仅包含一个容器且没有默认系统资源需求和限制的Pod对象：

```
~]$ kubectl run limit-pod1 --image=ikubernetes/myapp:v1 --restart=Never
```

Pod对象limit-pod1的详细信息中，容器状态信息段中，CPU资源被默认设定为如下配置，这正好符合了LimitRange对象中定义的默认值：

```
Limits:
  cpu: 1
Requests:
  cpu: 1
```

若Pod对象设定的系统资源需求量小于LimitRange中的最小用量限制，则会触发LimitRanger准入控制器拒绝相关的请求：

```
~]$ kubectl run limit-pod2 --image=ikubernetes/myapp:v1 \
  --restart=Never --requests='cpu=400m'
Error from server (Forbidden): pods "limit-pod4" is forbidden: minimum cpu usage
per Container is 500m, but request is 400m.
```

若Pod对象设定的系统资源限制量大于LimitRange中的最大用量限制，则一样会触发LimitRanger准入控制器拒绝相关的请求：

```
~]$ kubectl run limit-pod3 --image=ikubernetes/myapp:v1 --restart=Never --
limits=
  'cpu=3000m'
Error from server (Forbidden): pods "limit-pod2" is forbidden: maximum cpu
usage per Container is 2, but limit is 3.
```

事实上，在LimitRange对象中设置的默认的资源需求和资源限制，同最小资源用量及最大资源用量限制能够组合出多种不同的情形，不同组合场景下真正生效的结果也会存在不小的差异。另外，内存资源及PVC资源限制的实现与CPU资源大同小异，鉴于篇幅有限，这部分就留待读者自行验证了。

10.6.2 ResourceQuota资源与准入控制器

尽管LimitRange资源能限制单个容器、Pod及PVC等相关计算资源或存储资源的用量，但用户依然可以创建数量众多的此类资源对象进而侵占所有的系统资源。于是，Kubernetes提供了ResourceQuota资源用于定义名称空间的对象数量或系统资源配额，它支持限制每种资源类型的对象总数，以及所有对象所能消耗的计算资源及存储资源总量等。ResourceQuota准入控制器负责观察传入的请求，并确保它没有违反相应名称空间中ResourceQuota对象定义的任何约束。

于是，管理员可为每个名称空间分别创建一个ResourceQuota对象，随后，用户在名称空间中创建资源对象，ResourceQuota准入控制器将跟踪使用情况以确保它不超过相应ResourceQuota对象中定义的系统资源限制。用户创建或更新资源的操作违反配额约束将导致请求失败，API Server以HTTP状态代码“403FORBIDDEN”作为响应，并显示一条消息以提示可能违反的约束。不过，在名称空间上启用了CPU和内存等系统资源的配额后，用户创建Pod对象时必须指定资源需求或资源限制，否则，会触发ResourceQuota准入控制器拒绝执行相应的操作。

ResourceQuota对象可限制指定名称空间中非终止状态的所有Pod对象的计算资源需求及计算资源限制总量。

- cpu或requests.cpu: CPU资源需求的总量限额。
- memory或requests.memory: 内存资源需求的总量限额。
- limits.cpu: CPU资源限制的总量限额。
- limits.memory: 内存资源限制的总量限额。

ResourceQuota对象支持限制特定名称空间中可以使用的PVC数量和这些PVC资源的空间大小总量，以及特定名称空间中可在指定的StorageClass上使用的PVC数量和这些PVC资源的总数：

- requests.storage: 所有PVC存储需求的总量限额。

- `persistentvolumeclaims`: 可以创建的PVC总数。

- `<s storage-class-name>.storageclass.storage.k8s.io/requests.storage`: 指定存储类上可使用的所有PVC存储需求的总量限额。

- `<s storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`: 指定存储类上可使用的PVC总数。

- `requests.ephemeral-storage`: 所有Pod可用的本地临时存储需求的总量。

- `limits.ephemeral-storage`: 所有Pod可用的本地临时存储限制的总量。

在1.9版本之前，ResourceQuota支持在名称空间级别的有限的几种资源集上设定对象计数配额，如Pods、Services和ConfigMAPs等，而自1.9版本起支持以“count/<resource>.<group>”的格式支持对所有资源类型对象的计数配额，如count/deployments.apps、count/deployments.extensions和count/services等。

下面的配置清单示例定义了一个ResourceQuota资源对象，它配置了计算资源、存储资源及对象计数几个维度的限额：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-example
spec:
  hard:
    pods: "5"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    count/deployments.apps: "1"
    count/deployments.extensions: "1"
    persistentvolumeclaims: "2"
```

创建完成后，`describe`命令可以打印其限额的生效情况，例如，将上面的ResourceQuota对象创建于test名称空间中，其打印结果如下所示：

```
~]$ kubectl describe quota quota-example -n test
Name:                quota-example
Namespace:           test
Resource              Used    Hard
-----
count/deployments.apps    0      1
count/deployments.extensions 0      1
limits.cpu                0      2
limits.memory              0      2Gi
persistentvolumeclaims    0      2
pods                      0      5
requests.cpu               0      1
requests.memory            0      1Gi
```

在test名称空间中创建Deployment等对象即可进行配额测试，例如，下面的命令创建了一个有着3个Pod副本的Deployment对象myapp-deploy:

```
~]$ kubectl run myapp-deploy --image=ikubernetes/myapp:v1 --replicas=3 \
  --namespace=test --requests='cpu=200m,memory=256Mi' --
  limits='cpu=500m,memory=256Mi'
```

创建完成后，test名称空间上的ResourceQuota对象quota-example的各配置属性也相应地变成了如下状态:

Resource	Used	Hard
-----	----	----
count/deployments.apps	1	2
count/deployments.extensions	1	2
limits.cpu	1200m	2
limits.memory	768Mi	2Gi
persistentvolumeclaims	0	2
pods	3	5
requests.cpu	500m	1
requests.memory	640Mi	1Gi

此时，再扩展myapp-deploy的规模则会很快遇到某一项配额的限制而导致扩展受阻。由分析可知，将Pod副本数量扩展至5个就会达到limits.cpu资源上限而导致第5个扩展失败。读者可自行扩展并测试其结果。

需要注意的是，资源配额仅对那些在ResourceQuota对象创建之后生成的对象有效，对已经存在的对象不会产生任何限制。而且，一旦启用了计算资源需求和计算资源限制配额，那么创建的任何Pod对象

都必须设置此两类属性，否则Pod对象的创建将会被相应的ResourceQuota对象所阻止。无须手动为每个Pod对象设置此两类属性时，可以使用LimitRange对象为其设置默认值。

每个ResourceQuota对象上还支持定义一组适用范围（scope），用于定义其配额仅生效于这组适用范围交集内的对象，目前可用的适用范围包括Terminating、NotTerminating、BestEffort和NotBestEffort，具体说明如下。

- Terminating: 匹配.spec.activeDeadlineSeconds的属性值大于等于0的所有Pod对象。

- NotTerminating: 匹配.spec.activeDeadlineSeconds的属性值为空的所有Pod对象。

- BestEffort: 匹配所有位于BestEffort QoS类别的Pod对象。

- NotBestEffort: 匹配所有非BestEffort QoS类别的Pod对象。

另外，自1.8版本起，管理员即可以设置不同的优先级类别（PriorityClass）来创建Pod对象，而自1.11版本起，Kubernetes开始支持对每个PriorityClass对象分别设定资源限额，管理员可以使用scopeSelector字段，从而根据Pod对象的优先级控制Pod资源对系统资源的消耗。

10.6.3 PodSecurityPolicy

PodSecurityPolicy（简称**PSP**）是集群级别的资源类型，用于控制用户在配置**Pod**资源的期望状态时可以设定的特权类的属性，如是否可以使用特权容器、根命名空间和主机文件系统，以及可使用的主机网络和端口、卷类型和**Linux Capabilities**等。不过，**PSP**对象定义的策略本身并不会直接发生作用，它们需要经由**PodSecurityPolicy**准入控制器检查并强制生效。

不过，**PSP**准入控制器默认是处于未启用状态的，原因是在未创建任何**PSP**对象的情况下启用此准入控制器将阻止在集群中创建任何**Pod**对象。不过，**PSP**资源的API接口

（`policy/v1beta1/podsecuritypolicy`）独立于**PSP**准入控制器，因此管理员可以事先定义好所需要的**Pod**安全策略，而后再设置**kube-apiserver**启用**PSP**准入控制器。不当的**Pod**安全策略可能会产生难以预料的副作用，因此请确保所添加的任何**PSP**对象都经过了充分的测试。

启用**PSP**准入控制器后若要部署任何**Pod**对象，则相关的**User Account**及**Service Account**必须全部获得了恰当的**Pod**安全策略授权。以常规用户的身份直接创建**Pod**对象时，**PSP**准入控制器将根据账户有权使用的**Pod**安全策略验证其凭据。若不存在任何策略支持**Pod**对象安全性要求，则会拒绝相关的创建操作。而基于控制器（如**Deployment**）创建**Pod**对象时，**Kubernetes**会根据服务账户有权使用的**Pod**安全策略验证**Pod**的**Service Account**凭据。若没有策略支持**Pod**对象的安全性要求，则**Pod**控制器自身能够成功创建，但**Pod**对象不会。

事实上，即便是在启用了**PSP**准入控制器的情况下创建的**PSP**对象也依然不会发生效用，而是要由授权插件（如**RBAC**）将“**use**”操作权限授权给特定的**Role**或**ClusterRole**，而后将**User Account**或**Service Account**完成角色绑定才可以。下面简单说明一下设定重要的**Pod**安全策略，而后启用**PSP**准入控制器使其生效的方法。

1. 设置特权及受限的**PSP**对象

一般说来，`system: masters`组内的管理员用户、`system: node`组内的**kubelet**，以及**kube-system**名称空间中的**Service Account**需要拥有

创建各类Pod对象的授权，包括特权Pod对象。因此，首先要创建一个特权PSP，授予相关的管理员权限。一个示例性的配置清单如下：

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'
```

相应地，经过API Server成功认证的User Account或服务Account多数都应该具有创建非特权Pod对象的授权，因此，它们应该获取受限的安全策略。下面的配置清单定义了一个非特权的安全策略，它禁止了大多数的特权操作：

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName: 'docker/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName: 'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
  - ALL
```

```

# Allow core volume types.
volumes:
  - 'configMap'
  - 'emptyDir'
  - 'projected'
  - 'secret'
  - 'downwardAPI'
  # Assume that persistentVolumes set up by the cluster admin are safe to use.
  - 'persistentVolumeClaim'
hostNetwork: false
hostIPC: false
hostPID: false
runAsUser:
  # Require the container to run without root privileges.
  rule: 'MustRunAsNonRoot'
seLinux:
  # This policy assumes the nodes are using AppArmor rather than SELinux.
  rule: 'RunAsAny'
supplementalGroups:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
fsGroup:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
readOnlyRootFilesystem: false

```

将上述两个配置清单中定义的PSP资源提交并创建于API Server之上，随后便可授权特定的Role或ClusterRole资源通过use进行调用了。

2.创建ClusterRole并完成账户绑定

创建一个ClusterRole对象psp: privileged，授权其可以使用名为privileged的PSP对象，并创建一个ClusterRole对象psp: restricted，授权其可以使用名为restricted的PSP对象，如下面的配置清单所示：

```

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp:restricted
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames:
  - restricted
---
kind: ClusterRole

```

```
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp:privileged
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames:
  - privileged
```

而后创建ClusterRoleBinding对象，将system: masters、system: node和system: serviceaccounts: kube-system组的账户绑定至psp: privileged，让它们拥有管理员权限，并将system: authenticated组内的账户绑定至psp: restricted，授予它们创建普通Pod对象的权限：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: privileged-psp-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp:privileged
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:masters
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:node
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:serviceaccounts:kube-system
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: restricted-psp-user
roleRef:
  kind: ClusterRole
  name: psp:restricted
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:authenticated
```

3.启用PSP准入控制器

待上述步骤完成之后，设置kube-apiserver的--enable-admission-plugins选项，在其列表中添加“PodSecurityPolicy”项目，并重启kube-

apiserver便能启用PSP准入控制器。而对于使用kubeadm部署的Kubernetes集群来说，编辑Master节点上的/etc/kubernetes/manifests/kube-apiserver.yaml配置清单，在其--enable-admission-plugins选项后的列表上添加“PodSecurity- Policy”项目，等kube-apiserver相关的Pod对象自动重构完成之后即会启用PSP准入控制器。

随后，创建拥有特权securityContext的Pod资源配置清单，并以经过认证的User Account或服务Account分别创建特权和非特权的Pod对象便能验证Pod安全策略的效果。例如，为前面章节中创建的kube-user1使用Rolebinding对象在test名称空间绑定至名为admin的ClusterRole对象，授予其test名称空间的管理员权限，而后再由其尝试创建下面清单中定义的特权Pod对象：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-securitycontext
spec:
  containers:
  - name: busybox
    image: busybox
    imagePullPolicy: IfNotPresent
    command: ["/bin/sh", "-c", "sleep 86400"]
    securityContext:
      privileged: true
```

将上面的配置清单保存于配置文件中，如pod-test.yaml，而后进行创建测试：

```
~]$ kubectl apply -f pod-test.yaml -n test
Error from server (Forbidden): error when creating "pod-test.yaml": pods "pod-with-securitycontext" is forbidden: unable to validate against any pod security policy:
[spec.containers[0].securityContext.privileged: Invalid value: true: Privileged containers are not allowed]
```

命令结果显示创建操作被PSP定义的策略所拒绝，因为kube-user1用户被归类到system: authenticated组中，因此被关联到了restricted安全策略上。移除配置清单中的security-Context及其嵌套的字段再次执行创建操作即可成功完成，感兴趣的读者可自行测试。另外，读者可

按此方式授权特定的用户拥有特定类型的**Pod**对象创建权限，但策略冲突时可能会导致意料不到的结果，因此任何**Pod**安全策略在生效之前请务必做到充分测试。

10.7 本章小结

本章主要讲解了Kubernetes的认证、授权及准入控制相关的话题，其中重点说明了以下内容。

- Kubernetes系统的认证、授权及准入控制插件的工作流程。
- Service Account资源及其应用方式。
- HTTPS客户端证书认证及其在kubectl和tls bootstrap中的应用。
- Role及RoleBinding的工作机制及应用方式。
- ClusterRole及ClusterRoleBinding的工作机制及其应用方式。
- 借助默认的ClusterRole及ClusterRoleBinding实现集群及名称空间级别权限的快速授予。
- Dashboard及其分级权限授予。
- 使用LimitRange资源限制名称空间下容器、Pod及PVC对象的系统资源限制。
- 使用ResourceQuota设置名称空间的总资源限制。
- 使用PodSecurityPolicy设置创建Pod对象的安全策略。

第11章 网络模型与网络策略

自Docker技术诞生以来，采用容器技术用于开发、测试甚至是生产环境的企业或组织与日俱增。然而，将容器技术应用于生产环境时如何确定合适的网络方案依然是亟待解决的最大问题，这也曾是主机虚拟化时代的著名难题之一，它不仅涉及了网络中各组件的互连互通，还需要将容器与不相关的其他容器进行有效隔离以确保其安全性。本章将主要讲述容器网络模型的进化、Kubernetes的网络模型、常用网络插件以及网络策略等相关的话题。

11.1 Kubernetes网络模型及CNI插件

Docker的传统网络模型在应用至日趋复杂的实际业务场景时必将导致复杂性的几何级数上升，由此，Kubernetes设计了一种网络模型，它要求所有容器都能够通过一个扁平的网络平面直接进行通信（在同一IP网络中），无论它们是否运行于集群中的同一节点。不过，在Kubernetes集群中，IP地址分配是以Pod对象为单位，而非容器，同一Pod内的所有容器共享同一网络名称空间。

11.1.1 Docker容器的网络模型

Docker容器网络的原始模型主要有三种：**Bridge**（桥接）、**Host**（主机）及**Container**（容器）。**Bridge**模型借助于虚拟网桥设备为容器建立网络连接，**Host**模型则设定容器直接共享使用节点主机的网络名称空间，而**Container**模型则是指多个容器共享同一个网络名称空间，从而彼此之间能够以本地通信的方式建立连接。

Docker守护进程首次启动时，它会在当前节点上创建一个名为docker0的桥设备，并默认配置其使用172.17.0.0/16网络，该网络是Bridge模型的一种实现，也是创建Docker容器时默认使用的网络模型。如图11-1所示，创建Docker容器时，默认有四种网络可供选择使用，从而表现出了四种不同类型的容器，具体如下。

- Closed container**（封闭式容器）：此类容器使用“None”网络，它们没有对外通信的网络接口，而是仅具有I/O接口，通常仅用于不需要网络的后端作业处理场景。

- Bridged container**（桥接式容器）：此类容器使用“Bridge”模型的网络，对于每个网络接口，容器引擎都会为每个容器创建一对（两个）虚拟以太网设备，一个配置为容器的接口设备，另一个则在节点主机上接入指定的虚拟网桥设备（默认为docker0）。

- Open container**（开放式容器）：此类容器使用“Host”模型的网络，它们共享使用Docker主机的网络及其接口。

- Joined container**（联盟式容器）：此类容器共享使用某个已存在的容器的网络名称空间，即共享并使用指定的容器的网络及其接口。

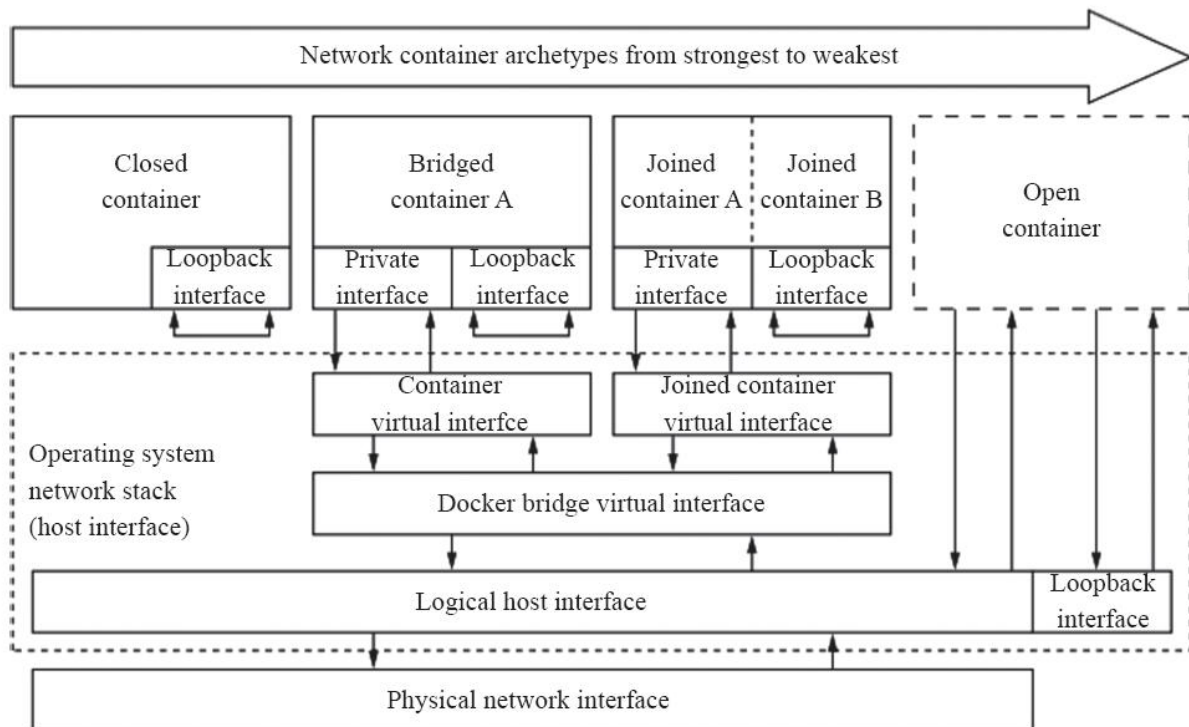


图11-1 Docker容器网络

上述的容器间通信仅描述了同一节点上的容器间通信的可用方案，这应该也是Docker设计者早期最为关注的容器通信目标。然而，在生产环境中使用容器技术时，跨节点的容器间通信反倒更为常见，可根据网络类型将其实现方式简单划分为如下几种。

- 为各Docker节点创建物理网络桥接接口，设定各节点上的容器使用此桥设备从而直接暴露于物理网络中。

- 配置各节点上的容器直接共享使用其节点的网络名称空间。

- 将容器接入指定的桥设备，如docker0，并设置其借助NAT机制进行通信。

第三种方案是较为流行且默认的解决方案。不过，此种方案的IPAM（IP Address Management）是基于容器主机本地范围进行的，每个节点上的容器都将从同一个网络172.17.0.0/16中获取IP地址，这就意味着不同Docker主机上的容器可能会使用相同的地址，因此它们也就无法直接通信。

解决此问题的通行方式是为NAT。所有接入到此桥设备上的容器均会被NAT隐藏，它们发往Docker主机外部的所有流量都会在执行过源地址转换后发出，并且默认是无法直接接收节点之外的其他主机发来的请求的。若要接入Docker主机外部的流量，则需要事先通过目标地址转换甚至额外的端口转换将其暴露于外部网络中，如图11-2所示。

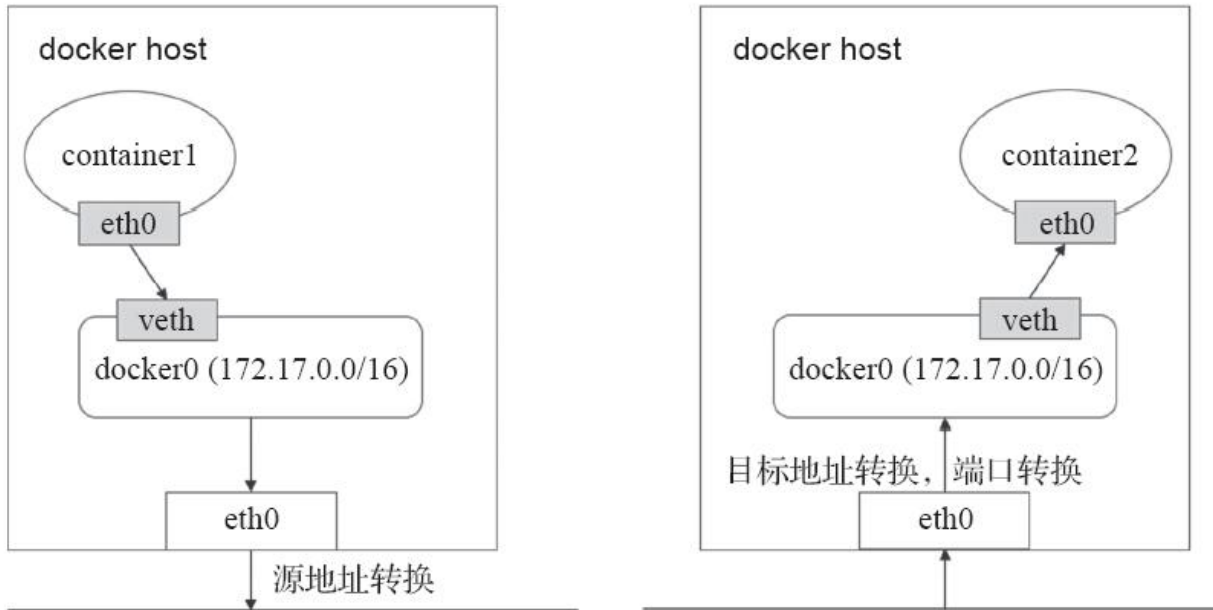


图11-2 跨节点容器间的通信示例图

故此，传统的解决方案中，多节点上的Docker容器间通信依赖于NAT机制转发实现。这种解决方案在网络规模庞大时将变得极为复杂、对系统资源的消耗较大且转发效率低下。此外，docker host的端口也是一种稀缺资源，静态分配和映射极易导致冲突，而动态分配又很容易导致模型的进一步复杂化。

事实上，Docker网络也可借助于第三方解决方案来规避NAT通信模型导致的复杂化问题，后来还发布了CNM（Container Network Model）规范，现在已经被Cisco Contiv、Kuryr、Open Virtual Networking（OVN）、Project Calico、VMware或Weave这些公司和项目所采纳。不过，这种模型被采用时，也就属于了容器编排的范畴。

11.1.2 Kubernetes网络模型

Kubernetes的网络模型主要可用于解决四类通信需求：同一Pod内容器间的通信（Container to Container）、Pod间的通信（Pod to Pod）、Service到Pod间的通信（Service to Pod）以及集群外部与Service之间的通信（external to Service）。

（1）容器间通信

Pod对象内的各容器共享同一网络名称空间，它通常由构建Pod对象的基础架构容器所提供，例如，由pause镜像启动的容器。所有运行于同一个Pod内的容器与同一主机上的多个进程类似，彼此之间可通过lo接口完成交互，如图11-3所示，Pod P内的Container1和Container2之间的通信即为容器间通信。

（2）Pod间通信

各Pod对象需要运行于同一个平面网络中，每个Pod对象拥有一个集群全局唯一的地址并可直接用于与其他Pod进行通信，如图11-3中的Pod P和Pod Q之间的通信。此网络也称为Pod网络。另外，运行Pod的各节点也会通过桥接设备等持有此平面网络中的一个IP地址，如图11-3中的cbr0接口，这就意味着Node到Pod间的通信也可在此网络上直接进行。因此，Pod间的通信或Pod到Node间的通信比较类似于同一IP网络中主机间进行的通信。

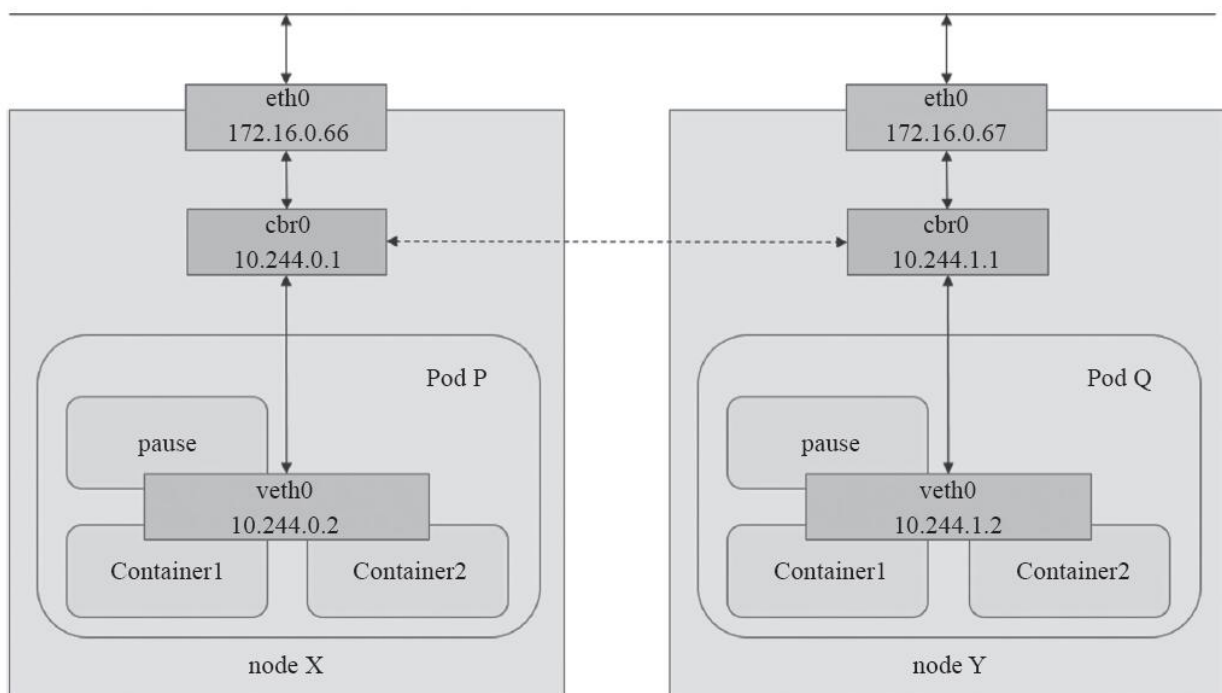


图11-3 Pod网络

此类通信模型中的通信需求也是Kubernetes的各网络插件需要着力解决的问题，它们的实现方式有叠加网络模型和路由网络模型等，流行的解决方案有十数种之多，例如前面使用到的flannel。

(3) Service与Pod间的通信

Service资源的专用网络也称为集群网络（Cluster Network），需要在启动kube-apiserver时经由“--service-cluster-ip-range”选项进行指定，如10.96.0.0/12，而每个Service对象在此网络中均拥有一个称为Cluster-IP的固定地址。管理员或用户对Service对象的创建或更改操作由API Server存储完成后触发各节点上的kube-proxy，并根据代理模式的不同将其定义为相应节点上的iptables规则或ipvs规则，借此完成从Service的Cluster-IP与Pod-IP之间的报文转发，如图11-4所示。

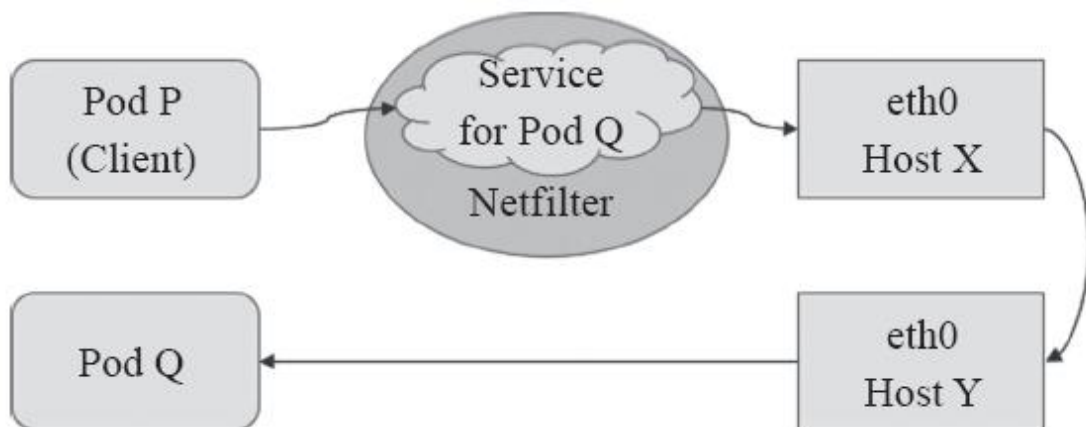


图11-4 Service和Pod

(4) 集群外部到Pod对象之间的通信

将集群外部的流量引入到Pod对象的方式有受限于Pod所在的工作节点范围的节点端口（nodePort）和主机网络（hostNetwork）两种，以及工作于集群级别的NodePort或LoadBalancer类型的Service对象。不过，即便是四层代理的模式也要经由两级转发才能到达目标Pod资源：请求流量首先到达外部负载均衡器，由其调度至某个工作节点之上，而后再由工作节点的netfilter（kube-proxy）组件上的规则（iptables或ipvs）调度至某个目标Pod对象。

以上4种通信方法的具体应用方式在前面的章节中已有详细描述，因此这里不再给出更进一步的说明。

11.1.3 Pod网络的实现方式

每个Pod对象内的基础架构容器均使用一个独立的网络名称空间，并共享给同一Pod内的其他容器使用。每个名称空间均有其专用的独立网络协议栈及其相关的网络接口设备。一个网络接口仅能属于一个网络名称空间，于是，运行多个Pod必然要求使用多个网络名称空间，也就需要用到多个网络接口设备。不过，一个易于实现的方案是使用软件实现的伪网络接口及模拟线缆将其连接至物理接口。伪网络接口的实现方案常见的有虚拟网桥、多路复用及硬件交换三种，如图11-5所示。

- 虚拟网桥：创建一对虚拟以太网接口（veth），一个接入容器内部，另一个留置于根名称空间内并借助于Linux内核桥接功能或OpenVSwitch（OVS）关联至真实的物理接口。

- 多路复用：多路复用可以由一个中间网络设备组成，它暴露了多个虚拟接口，可使用数据包转发规则来控制每个数据包转到的目标接口。MACVLAN为每个虚拟接口配置一个MAC地址并基于此地址完成二层报文收发，而IPVLAN是基于IP地址的并使用单个MAC，从而使其更适合VM。

- 硬件交换：现今市面上的大多数NIC都支持单根I/O虚拟化（SR-IOV），它是创建虚拟设备的一种实现方式。每个虚拟设备自身均表现为一个独立的PCI设备，并有着自己的VLAN及与硬件强制关联的QoS。SR-IOV提供了接近硬件级别的性能，但在公共云中通常是不可用的。

大多数情况下，用户希望创建跨越多个L2或L3的逻辑网络子网，这就要借助于叠加封装协议来实现（最常见的是VXLAN，它将叠加流量封装到UDP数据包中）。不过，由于控制平面缺乏标准化，VXLAN可能会引入更高的开销，并且来自不同供应商的多个VXLAN网络通常无法互操作。而Kubernetes还将大量使用iptables和NAT来拦截进入逻辑/虚拟地址的流量并将其路由到适当的物理目的地。

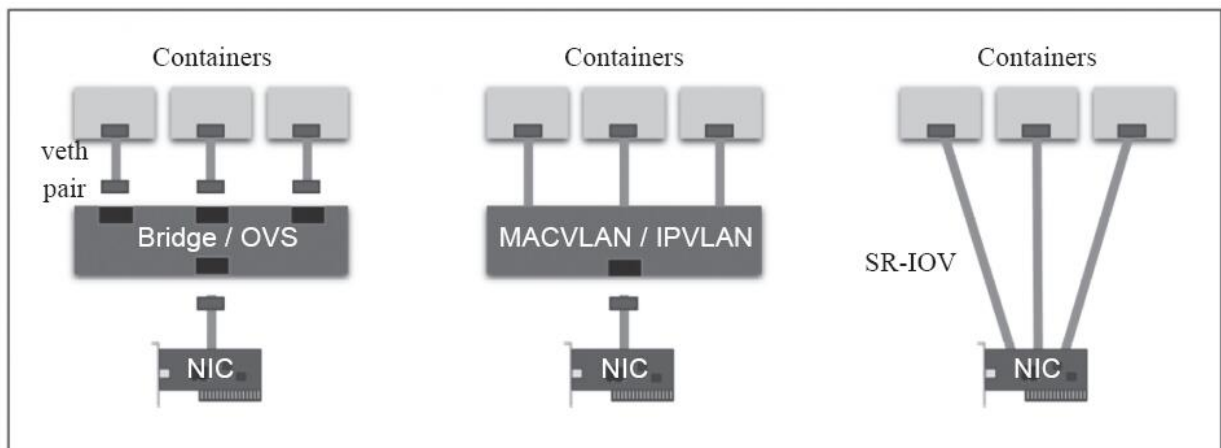


图11-5 虚拟网桥、多路复用及硬件交换

无论上述哪种方式应用于容器环境中，其实现过程都需要大量的操作步骤。不过，目前Kubernetes支持使用CNI插件来编排网络，以实现Pod及集群网络管理功能的自动化。每次Pod被初始化或删除时，kubelet都会调用默认的CNI插件创建一个虚拟设备接口附加到相关的底层网络，为其设置IP地址、路由信息并将其映射到Pod对象的网络名称空间。

配置Pod的网络时，kubelet首先在默认的/etc/cni/net.d/目录中查找CNI JSON配置文件，接着基于type属性到/opt/cni/bin/中查找相关的插件二进制文件，如下面示例中的“portmap”。随后，由CNI插件调用IPAM插件（IP地址管理）来设置每个接口的IP地址，如host-local或dhcp等：

```
~]$ cat /etc/cni/net.d/10-flannel.conflist
{
  "name": "cbr0",
  "plugins": [
    {
      "type": "flannel",
      "delegate": {
        "hairpinMode": true,
        "isDefaultGateway": true
      }
    },
    {
      "type": "portmap",
      "capabilities": {
        "portMappings": true
      }
    }
  ]
}
```

kubelet基于包含命令参数CNI_ARGS、CNI_COMMAND、CNI_IFNAME、CNI_NETNS、CNI_CONTAINERID、CNI_PATH的环境变量调用CNI插件，并经由stdin流式传输json.conf文件。被调用的插件使用JSON格式的文本信息进行响应，描述操作结果和状态。借助于插件框架，有着熟练的Go编程能力的读者，可以轻松开发出自己的CNI插件，或者扩展现有插件。

配置网络接口时，kubelet将Pod对象的名称和名称空间作为CNI_ARGS变量的一部分进行传递

（如“K8S_POD_NAMESPACE=default; K8S_POD_NAME=myapp-6d9f48c5d9-n77qp; ”）。它可以定义每个Pod对象或Pod网络名称空间的网络配置（例如，将每个网络名称空间放在不同的子网中）。未来的Kubernetes版本将网络视为一等的公民，并将网络配置作为Pod对象或名称空间规范的一部分，就像内存、CPU和存储卷一样。目前，可以使用注解（annotations）来存储配置或记录Pod网络数据/状态。

11.1.4 CNI插件及其常见的实现

Kubernetes设计了网络模型，但将其实现交给了网络插件。于是，各种解决方案不断涌现。为了规范及兼容各种解决方案，CoreOS和Google联合制定了CNI（Container Network Interface）标准，旨在定义容器网络模型规范。它连接了两个组件：容器管理系统和网络插件。它们之间通过JSON格式的文件进行通信，以实现容器的网络功能。具体的工作均由插件来实现，包括创建容器netns、关联网络接口到对应的netns以及为网络接口分配IP等。CNI的基本思想是：容器运行时环境在创建容器时，先创建好网络名称空间（netns），然后调用CNI插件为这个netns配置网络，而后再启动容器内的进程。

CNI本身只是规范，付诸生产还需要有特定的实现。目前，CNI提供的插件分为三类：main、meta和ipam。main一类的插件主要在于实现某种特定的网络功能，例如loopback、bridge、macvlan和ipvlan等；meta一类的插件自身并不提供任何网络实现，而是用于调用其他插件，例如调用flannel；ipam仅用于分配IP地址，而不提供网络实现。

CNI具有很强的扩展性和灵活性，例如，如果用户对某个插件具有额外的需求，则可以通过输入中的args和环境变量CNI_ARGS进行传递，然后在插件中实现自定义的功能，这大大增加了它的扩展性。另外，CNI插件将main和ipam分开，赋予了用户自由组合它们的机制，甚至一个CNI插件也可以直接调用另外一个CNI插件。CNI目前已经是Kubernetes当前推荐的网络方案。常见的CNI网络插件包含如下这些主流的项目。

- Flannel：一个为Kubernetes提供叠加网络的网络插件，它基于Linux TUN/TAP，使用UDP封装IP报文来创建叠加网络，并借助etcd维护网络的分配情况。

- Calico：一个基于BGP的三层网络插件，并且也支持网络策略来实现网络的访问控制；它在每台机器上运行一个vRouter，利用Linux内核来转发网络数据包，并借助iptables实现防火墙等功能。

- Canal：由Flannel和Calico联合发布的一个统一网络插件，提供CNI网络插件，并支持网络策略。

·Weave Net: Weave Net是一个多主机容器的网络方案，支持去中心化的控制平面，在各个host上的wRouter间建立Full Mesh的TCP连接，并通过Gossip来同步控制信息。

·数据平面上，Weave通过UDP封装实现L2Overlay，封装支持两种模式，一种是运行在user space的sleeve mode，另一种是运行在kernal space的fastpath mode。

·Contiv: 思科开源的容器网络方案，主要提供基于Policy的网络管理，并与主流容器编排系统集成；Contiv最主要的优势是直接提供了多租户网络，并支持L2（VLAN）、L3（BGP）、Overlay（VXLAN）等。

·OpenContrail: Juniper推出的开源网络虚拟化平台，其商业版本为Contrail。其主要由控制器和vRouter组成，控制器提供虚拟网络的配置、控制和分析功能，vRouter则提供分布式路由，负责虚拟路由器、虚拟网络的建立及数据转发。

·Romana: 由Panic Networks于2016年释出的开源项目，旨在借鉴路由汇聚（route aggregation）的思路来解决叠加方案为网络带来的开销。

·NSX-T: 由VMware提供，用于定义敏捷SDI（Software-Defined Infrastructure）以构建云原生应用环境；其旨在合并异构端点或技术栈的应用框架和架构，如vSphere、KVM、容器和bare metal等。

·kube-router: kube-router是Kubernetes网络的一体化解决方案，它可取代kube-proxy实现基于ipvs的Service，能为Pod提供网络，支持网络策略以及拥有完美兼容BGP的高级特性。

上述的CNI网络插件在实现方式、传输性能、功能特性等方面存在着不小的差别，部署时，用户需要根据网络环境和业务需要等来选择合适的项目。不过，就目前的统计（<https://thenewstack.io>）来看，flannel和calico两个项目是最为流行的选择，如图11-6所示。

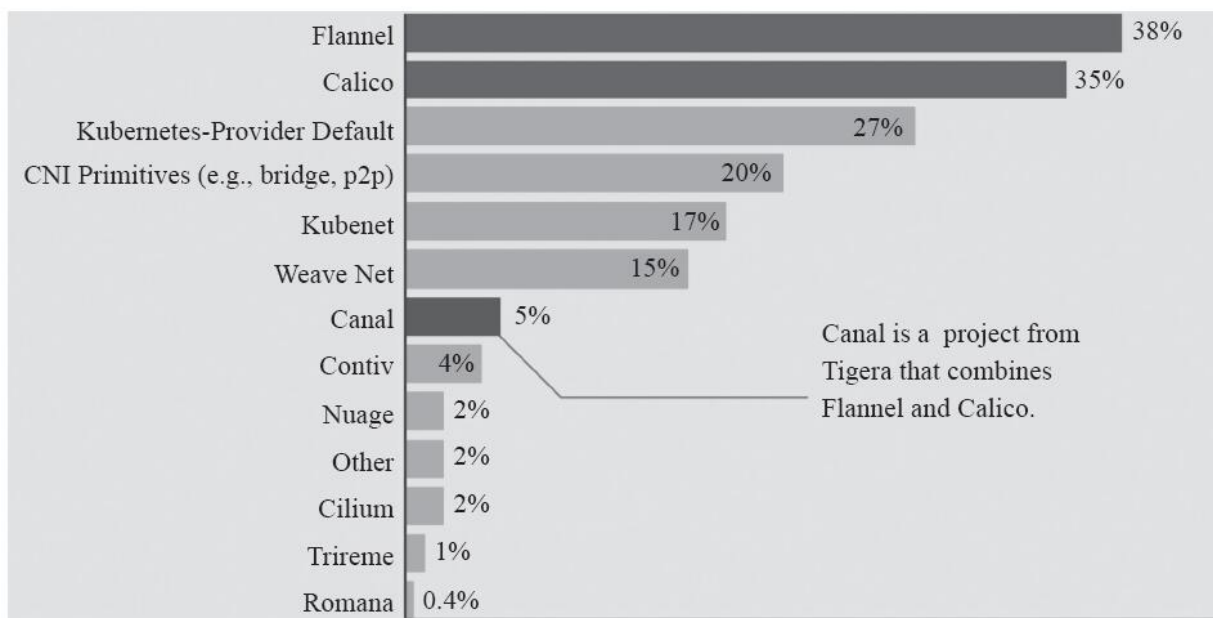


图11-6 CNI网络插件的采用率统计情况（含多选）

随着Kubernetes的演进，必将涌现出越来越多的CNI插件，它们各具特色，各有优劣。实践中，用户根据实际需要选择合用的方案即可。本章将介绍flannel和calico两种主流的方案及其部署和应用，不过，另一个非常值得关注的解决方案是kube-router。

11.2 flannel网络插件

各Docker主机在docker0桥上默认使用同一个子网，不同节点的容器很可能会得到相同的地址，于是跨节点的容器间通信会面临地址冲突的问题。另外，即使人为地设定多个节点上的docker0桥使用不同的子网，其报文也会因为在网络中缺乏路由信息而无法准确送达。事实上，各种CNI插件都至少要解决这两类问题。

对于第一个问题，flannel的解决办法是，预留使用一个网络，如10.244.0.0/16，而后自动为每个节点的Docker容器引擎分配一个子网，如10.244.1.0/24和10.244.20/24，并将其分配信息保存于etcd持久存储。对于第二个问题，flannel有着多种不同的处理方法，每一种处理方法也可以称为一种网络模型，或者称为flannel使用的后端。

·VxLAN: Linux内核自3.7.0版本起支持VxLAN，flannel的此种后端意味着使用内核中的VxLAN模块封装报文，这也是flannel较为推荐使用的方式。

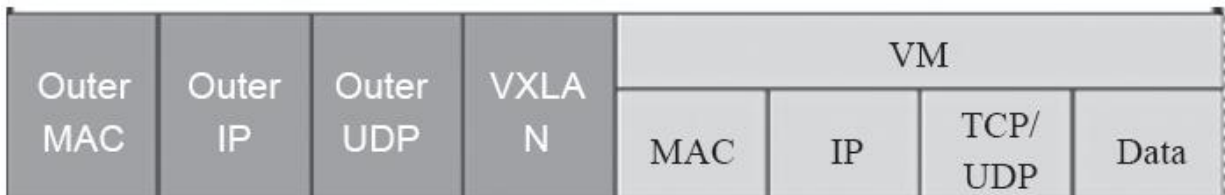


图11-7 VxLAN协议报文

·host-gw: 即Host GateWay，它通过在节点上创建到达目标容器地址的路由直接完成报文转发，因此这种方式要求各节点本身必须在同一个二层网络中，故该方式不太适用于较大的网络规模（大二层网络除外）。host-gw有着较好的转发性能，且易于设定，推荐对报文转发性能要求较高的场景使用。

·UDP: 使用普通UDP报文封装完成隧道转发，其性能较前两种方式要低很多，仅应该在不支持前两种方式的环境中使用。

flannel初创之后的一段时期内，不少环境中的Linux发行版的内核尚且不支持VxLAN，而host-gw模式有着略高的网络技术门槛，故此大多数部署场景只好使用UDP模式，flannel因而不幸地落下性能不好的声

名。不过，目前flannel的部署默认后端已经是叠加网络模型VxLAN。另外，除了这三种后端之外，flannel还实验性地支持AliVPC、AWS VPC、Alloc和GCE几种后端。

11.2.1 flannel的配置参数

为了跟踪各子网分配信息等，flannel使用etcd来存储虚拟IP和主机IP之间的映射，各个节点上运行的flanneld守护进程负责监视etcd中的信息并完成报文路由。默认情况下，flannel的配置信息保存于etcd的键名/coreos.com/network/config之下，可以使用etcd服务的客户端工具来设定或修改其可用的相关配置。config的值是一个JSON格式的字典数据结构，它可以使用的键包含以下几个。

1) **Network**: flannel于全局使用的CIDR格式的IPv4网络，字符串格式，此为必选键，余下的均为可选。

2) **SubnetLen**: 将Network属性指定的IPv4网络基于指定位的掩码切割为供各节点使用的子网，此网络的掩码小于24时（如16），其切割子网时使用的掩码默认为24位。

3) **SubnetMin**: 可用作分配给节点使用的起始子网，默认为切分完成后的第一个子网；字符串格式。

4) **SubnetMax**: 可用作分配给节点使用的最大子网，默认为切分完成后最大的一个子网；字符串格式。

5) **Backend**: flannel要使用的后端类型，以及后端的相关配置，字典格式；VxLAN、host-gw和UDP后端各有其相关的参数。

例如下面的配置示例中，全局网络为“10.244.0.0/16”，切分子网时用到的掩码长度为24，将相应的子网10.244.0.0/24-10.244.255.0/24分别分配给每一个工作节点使用，选择VxLAN作为使用的后端类型，并监听于8472端口：

```
{
  "Network": "10.244.0.0/16",
  "SubnetLen": 24,
  "Backend": {
    "Type": "VxLAN",
    "Port": 8472
  }
}
```

以上配置信息可直接由**flannel**保存于**etcd**存储中，也可交由**Kubernetes**进行存储。具体使用的方式取决于管理员或部署程序的默认配置。另外，**flannel**默认使用**VxLAN**后端，但**VxLAN direct routing**和**host-gw**却有着更好的性能表现。

另外，它将通过名为kube-flannel-ds的DaemonSet控制器资源，在每个节点运行一个flannel相关的Pod对象。DaemonSet控制器的Pod模板中使用“hostNetwork: true”配置每个节点上的Pod资源直接共享使用节点的网络名称空间以完成网络配置，其配置结果直接生效于节点的根网络名称空间。

传统的VxLAN后端使用隧道网络转发叠加网络的通信报文会导致不少的流量开销，于是flannel的VxLAN后端还支持DirectRouting模式，它通过添加必要的路由信息使用节点的二层网络直接发送Pod的通信报文，仅在跨IP网络时，才启用传统的隧道方式转发通信流量。由于大部分场景中都省去了隧道首部开销，因此DirectRouting通信模式的性能基本接近于直接使用二层物理网络，其架构模型如图11-9所示。

将flannel项目官方提供的配置清单下载至本地，如存储为Master节点上的/etc/kubernetes/manifests/kube-flannel.yaml文件，而后将其ConfigMap资源kube-flannel-cfg的data字段中的网络配置部分修改为如下内容所示，并使用“kubectl apply”命令重新应用于集群中即可：

```
net-conf.json: |
  {
    "Network": "10.244.0.0/16",
    "Backend": {
      "Type": "VxLAN",
      "Directrouting": true
    }
  }
```

配置完成后，在每个节点上执行路由查看命令“ip route show”可以看到其生成的路由规则，如下所示的结果是在本书的部署示例拓扑环境中的node01上生成的路由信息，其中，10.244.1.0/24网络位于本机上（node01节点），其他的目标网络则分别位于集群中的每个主机之上，包括master节点：

```
10.244.0.0/24 via 172.16.0.70 dev ens33
10.244.1.0/24 dev cni0 proto kernel scope link src 10.244.1.1
10.244.2.0/24 via 172.16.0.67 dev ens33
10.244.3.0/24 via 172.16.0.68 dev ens33
```

此时，在各个集群节点上执行“iptables-nL”命令可以看到，iptables filter表的FORWARD链上由其生成了如下两条转发规则，它显式放行了

10.244.0.0/16网络进出的所有报文，用于确保由物理接口接收或发送的目标地址或源地址为10.244.0.0/16网络的所有报文均能够正常通行。这些是Direct Routing模式得以实现的必要条件：

target	prot	opt	source	destination
ACCEPT	all	--	10.244.0.0/16	0.0.0.0/0
ACCEPT	all	--	0.0.0.0/0	10.244.0.0/16

各个节点上依然存在flannel相关接口，原因是对于那些无法通过直接路由到达的主机上的Pod（非同一个二层网络），它依然是采用VxLAN的隧道转发机制。按需启动两个Pod测试其通信，并通过相关接口捕获通信报文即可分析其结果。



注意 为了保证所有Pod均能得到正确的网络配置，建议在创建Pod资源之前事先配置好网络插件，甚至是事先了解并根据自身业务需求测试完成中意的目标网络插件，在选型完成后再部署Kubernetes集群，而尽量避免中途修改，否则有些Pod资源可能会需要重建。

VxLAN Directrouting后端转发模式同时兼具了VxLAN后端和host-gw后端的优势，既保证了传输性能，又具备了跨二层网络转发报文的能力。

另外，VxLAN后端的可用配置参数除了Type之外还有如下几个，它们都有其默认值，在用户需要自定义参数时可显式给出相关的配置。

- Type: VxLAN，字符串。
- VNI: VxLAN的标识符，默认为1；数值型数据。
- Port: 用于发送封装的报文的UDP端口，默认为8472；数值型数据。
- GBP: 全称为Group Based Policy，配置是否启用VxLAN的基于组的策略机制，默认为否；布尔型数据。
- DirectRouting: 是否为同一个二层网络中的节点启用直接路由机制，类似于host-gw后端的功能；此种场景下，VxLAN仅用于为那些不

在同一个二层网络中的节点封装并转发报文；布尔型数据。

11.2.3 host-gw后端

host-gw后端通过添加必要的路由信息使用节点的二层网络直接发送Pod的通信报文，其工作方式类似于VxLAN后端中direct routing的功能，但不包括其VxLAN的隧道转发能力。其工作模型示意图如图11-10所示。

编辑kube-flannel配置清单，将ConfigMap资源kube-flannel-cfg的data字段中网络配置部分修改为如下所示的内容，并使用“kubectl apply”命令重新应用于集群中即可配置flannel使用host-gw后端：

```
net-conf.json: |
  {
    "Network": "10.244.0.0/16",
    "Backend": {
      "Type": "host-gw"
    }
  }
```

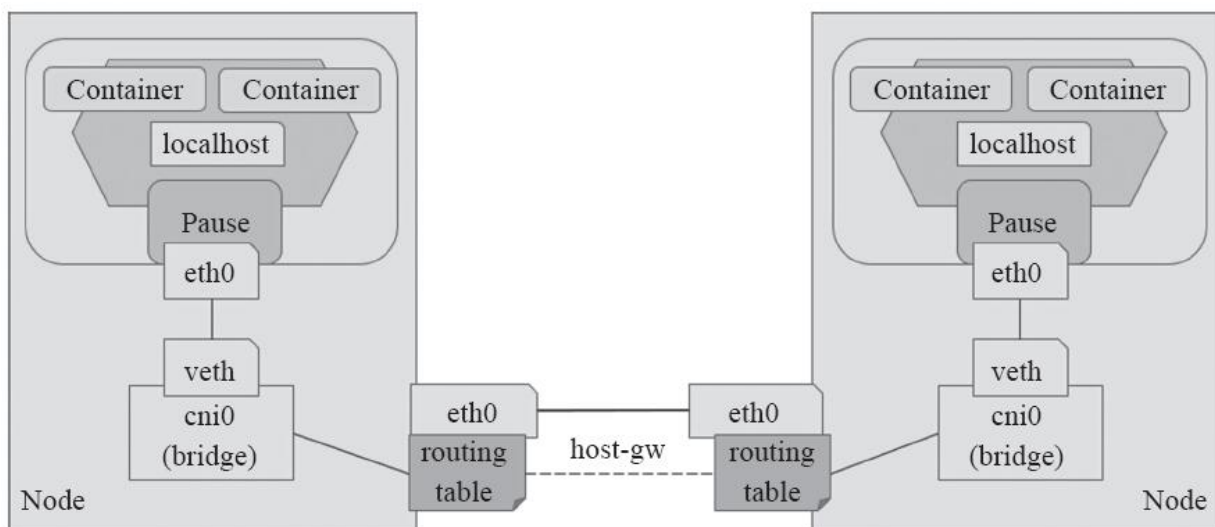


图11-10 host-gw后端

配置完成后，各节点会生成类似于VxLAN direct routing一样的路由及iptables规则以实现二层转发Pod网络的通信报文，省去了隧道转发模式的额外开销。不过，对于非同一个二层网络的报文的转发，host-gw则无能为力。

类似host-gw或VxLAN direct routing这种使用静态路由的方式来实现二层转发虽然较之VxLAN有着更低的资源开销和更好的性能表现，但在Kubernetes集群规模较大时其路由信息的规模也将变得庞大且不易维护。

此外，flannel自身并不具备为Pod网络实施网络策略以实现其网络通信隔离的能力，但它能够借助于Canal项目构建网络策略功能。

11.3 网络策略

网络策略（**Network Policy**）是用于控制分组的**Pod**资源彼此之间如何进行通信，以及分组的**Pod**资源如何与其他网络端点进行通信的规范。它用于为**Kubernetes**实现更为精细的流量控制，实现租户隔离机制。**Kubernetes**使用标准的资源对象“**NetworkPolicy**”供管理员按需定义网络访问控制策略。

11.3.1 网络策略概述

Kubernetes的网络策略功能由其所使用的网络插件实现，因此，仅在使用那些支持网络策略功能的网络插件时才能够配置网络策略，如Calico、Canal及kube-router等。每种解决方案各有其特定的网络策略实现方式，它们的实现或依赖于节点自身的功能，或借助于Hypervisor的特性，也可能是网络自身的功能。Calico的calico/kube-controllers即为Calico项目中用于将用户定义的网络策略予以实现的组件，它主要依赖于节点的iptables来实现访问控制功能，如图11-11所示。其他支持网络策略的插件也有类似的将网络策略加以实现的“策略控制器”或“策略引擎”。

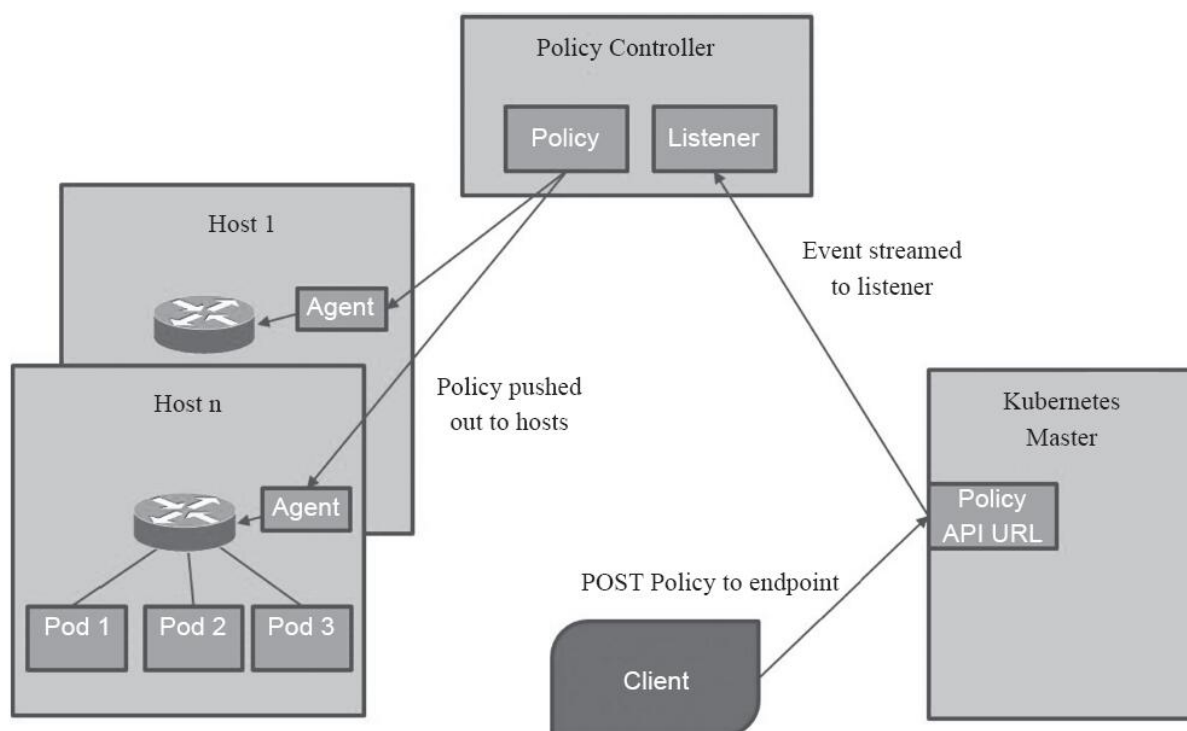


图11-11 网络策略组件构架

策略控制器用于监控创建Pod时所生成的新API端点，并按需为其附加网络策略。当发生需要配置策略的事件时，侦听器会监视到变化，控制器随即响应以进行接口配置和策略应用。

Pod的网络流量包含“流入”(Ingress)和“流出”(Egress)两种方向，每种方向的控制策略则包含“允许”和“禁止”两种。默认情况下，Pod处于非隔离状态，它们的流量可以自由来去。一旦有策略通过选择器规则将策略应用于Pod，那么所有未经明确允许的流量都将被网络策略拒绝，不过，其他未被选择器匹配到的Pod不受影响。



注意 Kubernetes自1.8版本起才支持Egress网络策略，此前的版本仅支持Ingress网络策略。

11.3.2 部署Canal提供网络策略功能

Canal代表了针对云原生应用程序的最佳策略网络解决方案，旨在让用户轻松地将Calico和flannel网络部署在一起作为统一的网络解决方案，将Calico的网络策略执行与Calico和flannel叠加以及非叠加网络连接选项的丰富功能相结合，如图11-12所示。

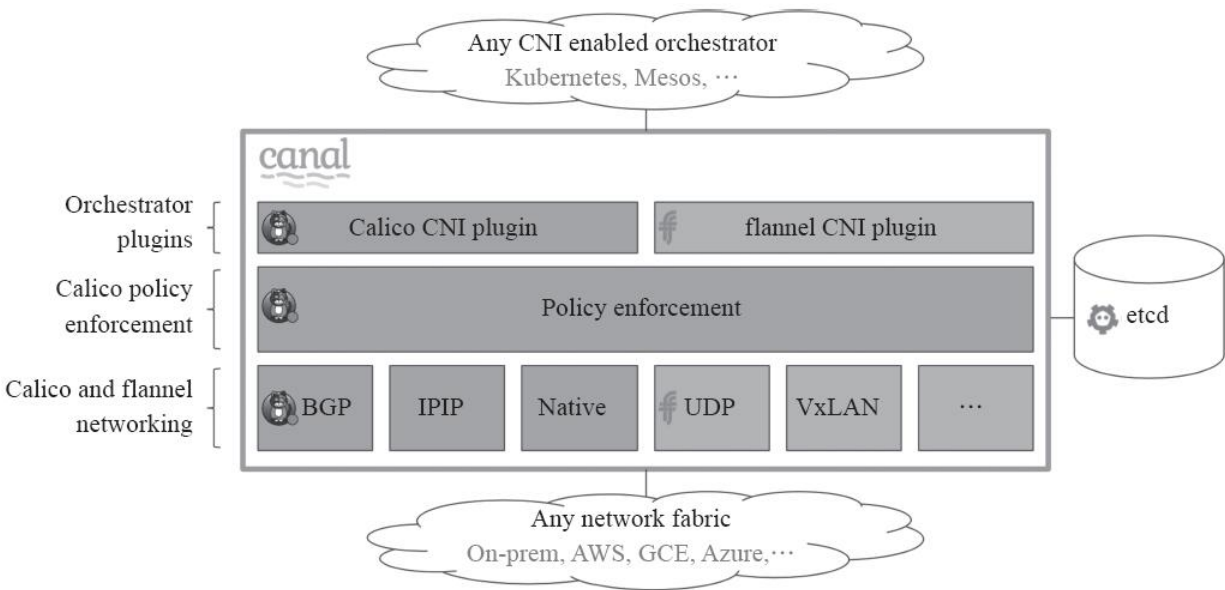


图11-12 Canal项目架构组件

换句话说，Calico项目既能够独立地为Kubernetes集群提供网络解决方案和网络策略，也能与flannel结合在一起，由flannel提供网络解决方案，而Calico此时仅用于提供网络策略，这时我们也可以将Calico称为Canal。Calico将数据存储于etcd中，它支持选择使用专用的etcd存储，也能够以Kubernetes API Server作为后端存储，这里选择以第二种方式进行。



注意 结合flannel工作时，Calico提供的默认配置清单中是以flannel默认使用的10.244.0.0/16为Pod网络，因此，请确保kube-controller-manager程序在启动时通过--cluster-cidr选项设置使用了此网络地址，并且--allocate-node-cidrs的值应设置为true。

部署之前，要在启用了RBAC的Kubernetes集群中设置必要的相关资源：

```
~]$ kubectl apply -f https://docs.projectcalico.org/v3.2/getting-started/kubernetes/installation/hosted/canal/rbac.yaml
```

接下来即可部署Canal提供网络策略：

```
~]$ kubectl apply -f https://docs.projectcalico.org/v3.2/getting-started/kubernetes/installation/hosted/canal/canal.yaml
```

需要注意的是，Canal目前直接使用Calico和flannel项目，代码本身并没有任何修改。因此，目前的Canal只是一种部署模式，用于安装和配置项目，从用户和编排系统的角度无缝地作为单一网络解决方案协同工作。未来，Canal项目可能会对Calico和flannel项目进行代码更改，实现安装和配置的进一步简化。

11.3.3 配置网络策略

Kubernetes系统中，报文流入和流出的核心组件是Pod资源，因此它们也是网络策略功能生效的主要目标，因此，NetworkPolicy对象也要使用标签选择器事先选择出一组Pod资源作为控制对象。一般来说，NetworkPolicy是定义在一组Pod资源上的用于管控入站流量的“Ingress规则”，或者说是管理出站流量的“Egress规则”，也可以是二者组合定义，而仅部分生效还是全部生效则需要由spec.policyTypes予以定义，如图11-13所示。

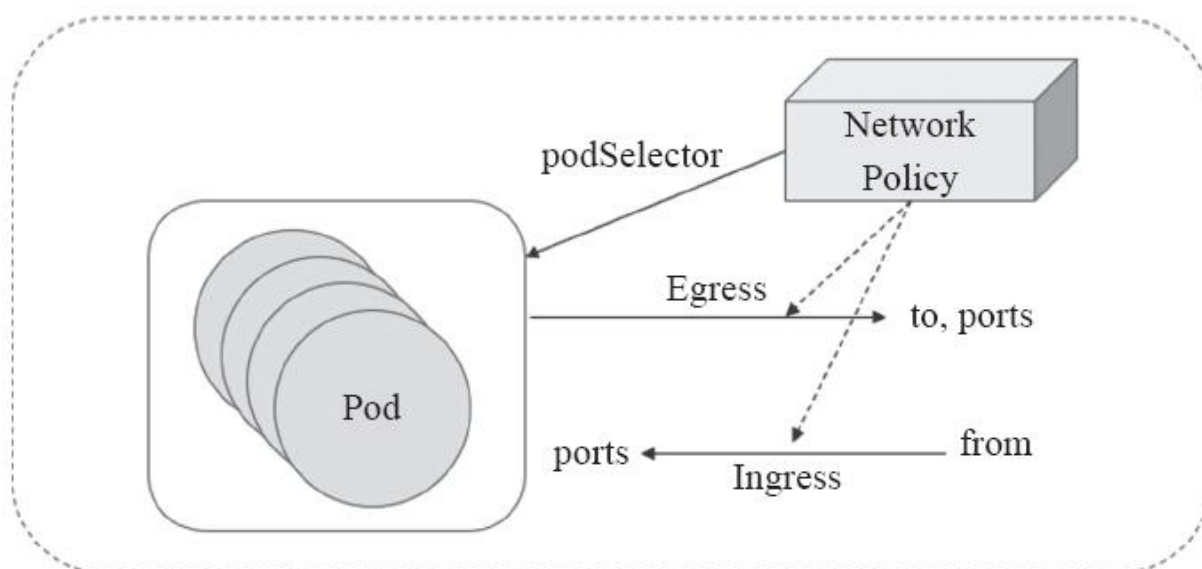


图11-13 网络策略示意图

默认情况下，Pod对象既可以接受来自任何来源的流量，也能够向外部发出期望的所有流量。而附加网络策略机制后，Pod对象会因NetworkPolicy对象的选定而被隔离：一旦名称空间中有任何NetworkPolicy对象匹配了某特定的Pod对象，则该Pod将拒绝NetworkPolicy所不允许的一切连接请求，而那些未被任何NetworkPolicy对象匹配到的其他Pod对象仍可接受所有流量。因此，就特定的Pod集合来说，入站和出站流量默认均处于放行状态，除非有规则能够明确匹配到它。然而，一旦在spec.policyTypes中指定了生效的规则类型，却在networkpolicy.spec字段中嵌套定义了没有任何规则的Ingress或Egress字段时，则表示拒绝相关方向上的一切流量。

定义网络策略时常用到的术语及说明具体如下。

- Pod组：由网络策略通过Pod选择器选定的一组Pod的集合，它们是规则生效的目标Pod；可由NetworkPolicy对象通过matchLabel或matchExpression选定。

- Egress：出站流量，即由特定的Pod组发往其他网络端点的流量，通常由流量的目标网络端点（to）和端口（ports）来进行定义。

- Ingress：入站流量，即由其他网络端点发往特定Pod组的流量，通常由流量发出的源站点（from）和流量的目标端口所定义。

- 端口（ports）：TCP或UDP的端口号。

- 端点（to, from）：流量目标和流量源相关的组件，它可以是CIDR格式的IP地址块（ipBlock）、网络名称空间选择器（namespaceSelector）匹配的名称空间，或Pod选择器（podSelector）匹配的Pod组。

无论是Ingress还是Egress流量，与选定的某Pod组通信的另一方都可使用“网络端点”予以描述，它通常是某名称空间中的一个或一组Pod资源，由namespaceSelector选定名称空间后，经由ipBlock或podSelector进行指定。Pod集合的选定方式如图11-14所示。

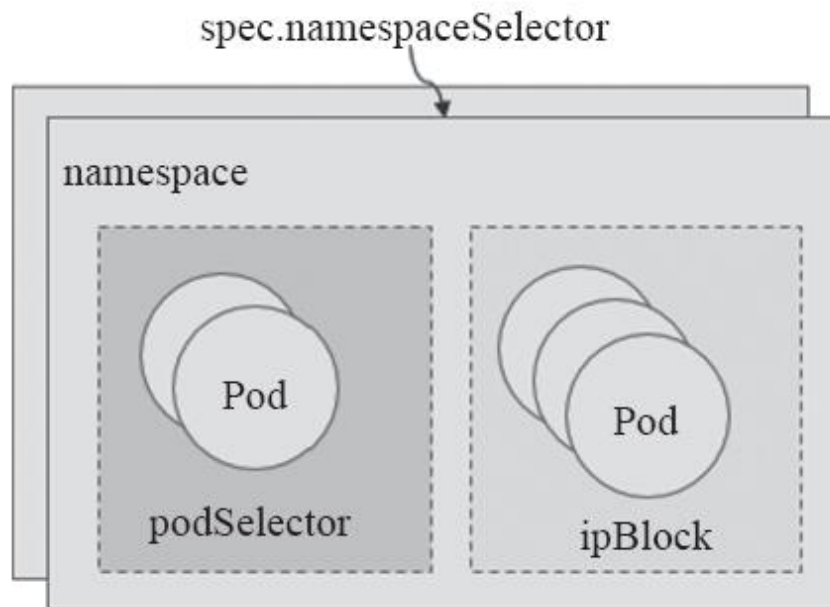


图11-14 Pod集合的选定方式

在Ingress规则中，网络端点也称为“源端点”，它们用from字段进行标识，而在Egress规则中，网络端点也称为“目标端点”，它们用to字段进行标识，如图11-13所示。不过，在未定义Ingress或Egress规则时，相关方向的流量均为“允许”，即默认为非隔离状态。而一旦在networkpolicy.spec中明确给出了Ingress或Egress字段，则它们的from或to字段的值就成了白名单列表，而空值意味着所有端点，即不限制访问。

11.3.4 管控入站流量

以提供服务为主要目的的Pod对象通常是请求流量的目标对象，但他们的服务未必应该为所有网络端点所访问，这就有必要对它们的访问许可施加控制。networkpolicy.spec中嵌套的Ingress字段用于定义入站流量规则，就特定的Pod集合来说，入站流量默认处于放行状态，除非在所有入站策略中，至少有一条规则能够明确匹配到它。Ingress字段的值是一个对象列表，它主要由以下两个字段组成。

·from<[]Object>: 可访问当前策略匹配到的Pod对象的源地址对象列表，多个项目之间的逻辑关系为“逻辑或”的关系；若未设置此字段或其值为空，则匹配一切源地址（默认的策略为不限制）；如果此字段至少有一个值，那么它将成为放行的源地址白名单，仅来源于此地址列表中的流量允许通过。

·ports<[]Object>: 当前策略匹配到的Pod集合的可被访问的端口对象列表，多个项目之间的逻辑关系为“逻辑或”的关系；若未设置此字段或其值为空，则匹配Pod集合上的所有端口（默认的策略为不限制）；如果此字段至少有一个值，那么它将成为允许被访问的Pod端口白名单列表，仅入站流量的目标端口处于此列表中方才准许通过。

需要注意的是，NetworkPolicy资源属于名称空间级别，它的有效作用范围为其所属的名称空间。

1. 设置默认的Ingress策略

必要时，用户可以创建一个NetworkPolicy来为名称空间设置一个“默认”的隔离策略，该策略选择所有的Pod对象，而后允许或拒绝任何到达这些Pod的入站流量。例如下面的策略示例，其通过policyTypes字段指明要生效Ingress类型的规则，但未定义任何Ingress字段，因此不能匹配到任一源端点，从而拒绝所有入站流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
spec:
```

```
podSelector: {}  
policyTypes: ["Ingress"]
```

若要将默认策略设置为允许所有的入站流量，则只需要显式定义**Ingress**字段，并将其值设置为空以匹配所有源端点即可，如下面示例中的定义。不过，没有为入站流量定义任何规则时，本身的默认规则即为允许访问，因此允许所有相关的入站流量时，本身无须定义默认规则，下面的示例只是为说明规则的定义格式：

```
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: allow-all-ingress  
spec:  
  podSelector: {}  
  policyTypes: ["Ingress"]  
  ingress:  
  - {}
```

实践中，通常将默认策略设置为拒绝所有的入站流量，而后显式放行允许的源端点的入站流量。

2.放行特定的入站流量

在**Ingress**规则中嵌套**from**和**ports**字段即可匹配特定的入站流量，仅定义**from**字段时将隐含本地**Pod**资源组的所有端口，而仅定义**ports**字段时则表示隐含所有的源端点。 **from**和**ports**同时定义时表示隐含“逻辑与”关系，它将匹配那些同时满足**from**和**ports**的定义的入站流量，即那些来自**from**指定的源端点，访问由当前**NetworkPolicy**的**podSelector**匹配的当前名称空间的**Pod**资源组上所指定的**ports**的请求，如图11-15所示。

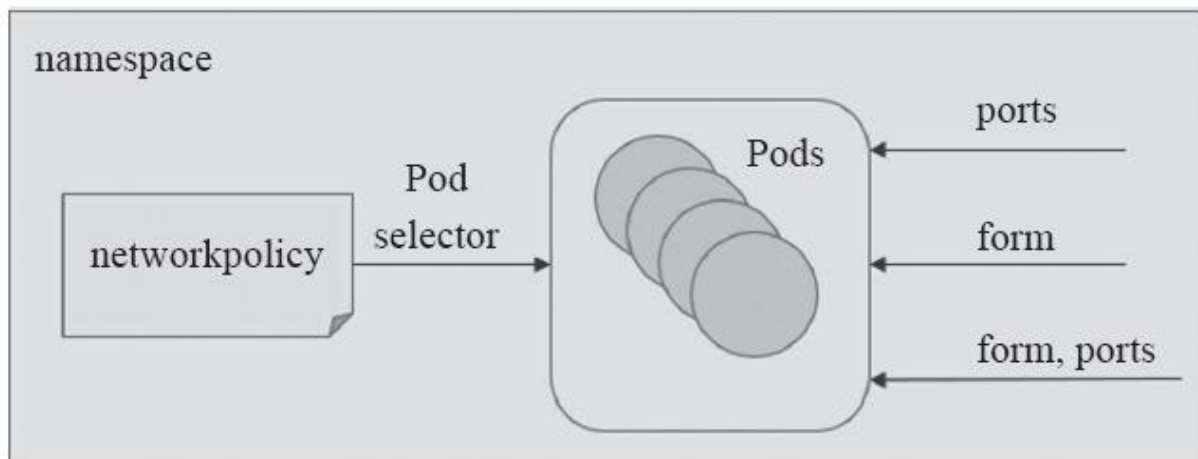


图11-15 Ingress规则的组成方式

`from`字段的值是一个对象列表，它可嵌套使用`ipBlock`、`namespaceSelector`和`podSelector`字段来定义流量来源，此三个字段匹配Pod资源的方式各有不同，同时使用两个或以上的字段，彼此之间隐含“逻辑或”关系。

- `ipBlock<Object>`：根据IP地址或网络地址块选择流量源端点。

- `namespaceSelector<Object>`：基于集群级别的标签挑选名称空间，它将匹配由此标签选择器选出的所有名称空间内的所有Pod对象；赋予字段以空值来表示挑选所有的名称空间，即源站点为所有名称空间内的所有Pod对象。

- `podSelector<Object>`：于NetworkPolicy所在的当前名称空间内基于标签选择器挑选Pod资源，赋予字段以空值来表示挑选当前名称空间内的所有Pod对象。

- `ports`字段的值也是一个对象列表，它嵌套`port`和`protocol`来定义流量的目标端口，即由NetworkPolicy匹配到的当前名称空间内的所有Pod资源上的端口。

- `port<string>`：端口号或在Container上定义的端口名称，未定义时匹配所有端口。

- `protocol<string>`：传输层协议的名称，TCP或UDP，默认为TCP。

下面配置清单中的网络策略示例定义了如何开放myapp pod资源给相应的源站点访问：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-myapp-ingress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes: ["Ingress"]
  ingress:
    - from:
      - ipBlock:
          cidr: 10.244.0.0/16
          except:
            - 10.244.3.0/24
      - podSelector:
          matchLabels:
            app: myapp
    ports:
      - protocol: TCP
        port: 80
```

它将default名称空间中拥有标签“app=myapp”的Pod资源的80/TCP端口开放给10.244.0.0/16网络内除10.244.3.0/24子网中的所有源端点，以及当前名称空间中拥有标签“app=myapp”的所有Pod资源访问，其他未匹配到的源端点的流量则取决于其他网络策略的定义，若没有任何匹配策略，则默认为允许访问。

11.3.5 管控出站流量

除非是仅于当前名称空间中即能完成所有的目标功能，否则，大多数情况下，一个名称空间中的Pod资源总是有对外请求的需求，如向CoreDNS请求解析名称等。因此，通常应该将出站流量的默认策略设置为准许通过。但如果有必要对其实施精细管理，仅放行那些有对外请求需要的Pod对象的出站流量，则也可先为名称空间设置“禁止所有”默认策略，而后定义明确的“准许”策略。

`networkpolicy.spec`中嵌套的Egress字段用于定义入站流量规则，就特定的Pod集合来说，出站流量一样默认处于放行状态，除非在所有入站策略中至少有一条规则能够明确匹配到它。Egress字段的值是一个字段列表，它主要由以下两个字段组成。

·`to<[]Object>`：由当前策略匹配到的Pod资源发起的出站流量的目标地址列表，多个项目之间为“或”（OR）关系；若未定义或字段值为空则意味着应用于所有目标地址（默认为不限制）；若明确给出了主机地址列表，则只有目标地址匹配列表中的主机地址的出站流量被放行。

·`ports<[]Object>`：出站流量的目标端口列表，多个端口之间为“或”（OR）关系；若未定义或字段值为空则意味着应用于所有端口（默认为不限制）；若明确给出了端口列表，则只有目标端口匹配列表中的端口的出站流量被放行。

Egress规则中，`to`和`ports`字段的值都是对象列表格式，它们可内嵌的字段分别与Ingress规则中的`from`和`ports`相同，区别仅是作用到的流量方向相反，如图11-16所示。

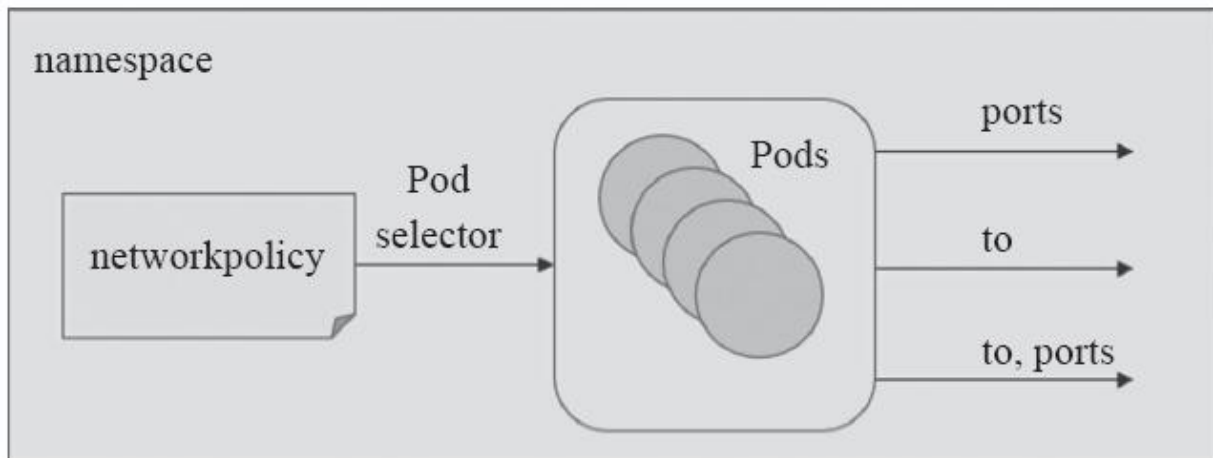


图11-16 Egress字段的组成方式

1. 设置默认Egress策略

类似于使用Ingress的使用方式，用户也可以通过创建一个NetworkPolicy对象来为名称空间设置一个默认的隔离策略，该策略选择所有的Pod对象，而后允许或拒绝由这些Pod发出的所有出站流量。例如下面的策略示例，它通过policyTypes字段指明要生效Egress类型的规则，但未定义任何Egress字段，因此不能匹配到任何目标端点，从而拒绝所有的入站流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-egress
spec:
  podSelector: {}
  policyTypes: ["Egress"]
```

实践中，需要进行严格隔离的环境通常将默认策略设置为拒绝所有出站流量，而后显式放行允许到达的目标端点的出站流量。

2. 放行特定的出站流量

下面的配置清单示例中定义了一个Egress规则，它对来自拥有“app=tomcat”的Pod对象的，到达标签为“app=nginx”的Pod对象的80端口，以及到达标签为“app=mysql”的Pod对象的3306端口的流量给予放行：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-tomcat-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: tomcat
  policyTypes: ["Egress"]
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: nginx
        ports:
          - protocol: TCP
            port: 80
    - to:
        - podSelector:
            matchLabels:
              app: mysql
        ports:
          - protocol: TCP
            port: 3306
```

需要注意的是，此配置清单中仅定义了出站规则，将入站流量的默认规则设置为拒绝所有时，还应该为具有标签“app=tomcat”的Pod对象放行入站流量。

11.3.6 隔离名称空间

实践中，通常需要彼此隔离所有的名称空间，但应该允许它们都能够与**kube-system**名称空间中的**Pod**资源进行流量交换，以实现监控和名称解析等各种管理功能。下面的配置清单示例为**default**名称空间定义了相关的规则，在出站和入站流量默认均为拒绝的情况下，它用于放行名称空间内部的各**Pod**对象之间的通信，以及与**kube-system**名称空间内各**Pod**间的通信：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: namespace-deny-all
  namespace: default
spec:
  policyTypes: ["Ingress", "Egress"]
  podSelector: {}
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: namespace-
  namespace: default
spec:
  policyTypes: ["Ingress", "Egress"]
  podSelector: {}
  ingress:
  - from:
    - namespaceSelector:
        matchExpressions:
        - key: name
          operator: In
          values: ["default", "kube-system"]
  egress:
  - to:
    - namespaceSelector:
        matchExpressions:
        - key: name
          operator: In
          values: ["default", "kube-system"]
```

需要注意的是，有些管理员可能会把一些系统附件部署到专有的名称空间中，例如把**Prometheus**监控系统部署到**prom**名称空间中等，所有这类的具有管理功能的附件所在的名称空间与每一个特定名称空间的出入流量都应该被放行。

11.3.7 网络策略应用案例

假设有名为testing的名称空间内运行着一组nginx Pod和一组myapp Pod。要求实现如下目标。

- 1) myapp Pod仅允许来自nginx Pod的流量访问其80/TCP端口，但可以向nginx Pod的所有端口发出出站流量。
- 2) nginx Pod允许任何源端点对其80/TCP端口的访问，并能够向任意端点发出出站流量。
- 3) myapp Pod和nginx Pod都可与kube-system名称空间的任意Pod进行任何类型的通信，以便于可以使用由kube-dns提供的名称解析服务等。

如图11-17所示，出站和入站的默认策略均为“禁止”。

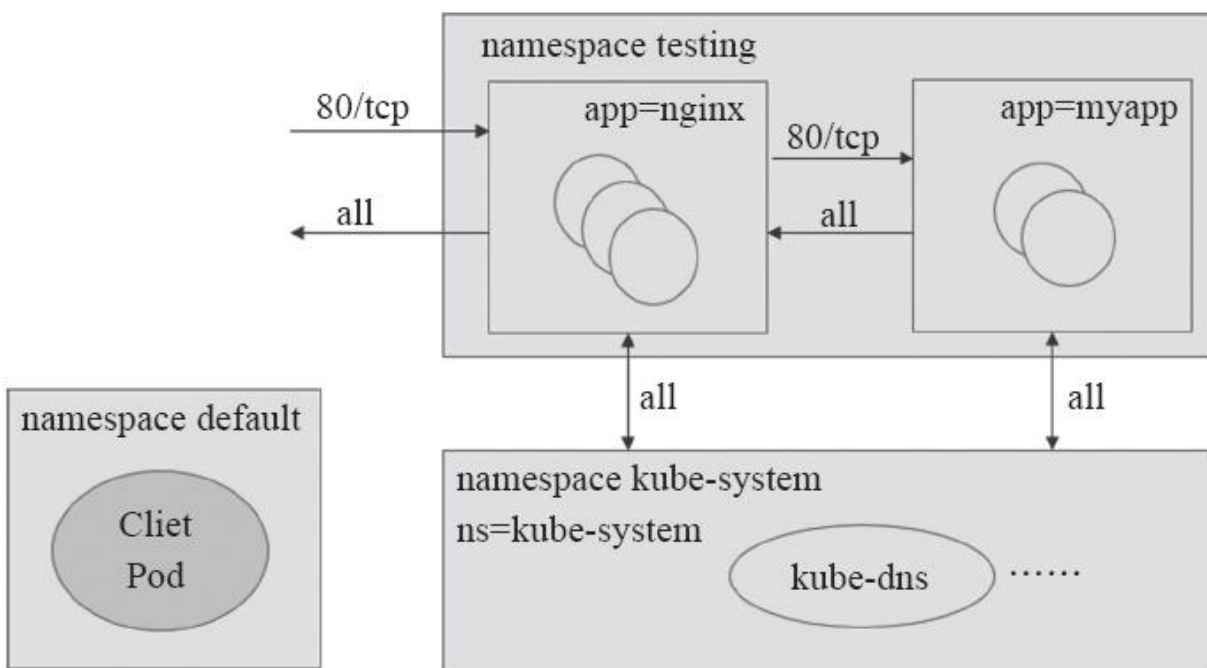


图11-17 案例拓扑

下面是测试实现步骤。

第一步： 创建testing名称空间，并于其内基于Deployment控制器创建用于测试用的nginx Pod和myapp Pod各一个，创建相关Pod资源时顺便为其创建与Deployment控制器同名的Service资源：

```
~]$ kubectl create namespace testing
~]$ kubectl run nginx --image=nginx:alpine --replicas=1 --namespace=testing \
  --port 80 --expose --labels app=nginx
~]$ kubectl run myapp --image=ikubernetes/myapp:v1 --replicas=1 \
  --namespace=testing --port 80 --expose --labels app=myapp
```

另外，为了便于在网络策略规则中引用kube-system名称空间，这里为其添加标签“ns=kube-system”：

```
~]$ kubectl label namespace kube-system ns=kube-system
```

待相关资源创建完成后，即可通过与其相关的Service资源的名称访问相关的服务。例如，另外启动一个终端，使用kubectl命令在default名称空间中创建一个用于测试的临时交互式客户端：

```
~]$ kubectl run cirros-$RANDOM --namespace=default --rm -it --image=cirros -- sh
/ #
```

而后基于此客户端分别测试访问nginx和myapp的服务，名称空间的默认网络策略为准许访问，接下来确认其访问请求可正常通过：

```
/ # curl nginx.testing
.....
<title>Welcome to nginx!</title>
.....
/ # curl myapp.testing
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
/ #
```

第二步： 定义网络策略清单文件（testing-netpol-denyall.yaml），将testing名称空间的入站及出站的默认策略修改为拒绝访问，并再一次进行访问测试：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
metadata:
  name: deny-all-traffic
  namespace: testing
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

接下来，将testing-netpol-denyall.yaml定义的默认策略deny-all-traffic予以应用，为testing名称空间设置默认的网络策略：

```
$ kubectl apply -f testing-netpol-denyall.yaml
```

回到第一步创建的交互式客户端，再次对nginx和myapp发起访问测试。为了避免长时间等待，这里为curl命令添加--connect-timeout选项为其定义连接超时时长。由下面的命令可知，此时无法再访问到nginx和myapp的相关服务：

```
/ # curl --connect-timeout 2 nginx.testing
curl: (28) connect() timed out!
/ # curl --connect-timeout 2 myapp.testing
curl: (28) connect() timed out!
/ #
```

第三步： 定义流量放行规则配置清单nginx-allow-all.yaml，放行nginx Pod之上80/TCP端口的所有流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: nginx-allow-all
  namespace: testing
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - ports:
    - port: 80
  - from:
    - namespaceSelector:
        matchLabels:
          ns: kube-system
  egress:
  - to:
  policyTypes:
```

- Ingress
 - Egress
-

将上述清单文件中定义的网络策略应用至集群中以创建相应的网络策略:

```
~]$ kubectl apply -f nginx-allow-all.yaml
```

而后再回到交互式测试客户端发起访问请求进行测试, 由下面的命令结果可知, **nginx**已经能够正常访问, 这同时也意味着由**kube-system**名称空间中的**kube-dns**进行的名称解析服务也为可用状态:

```
/ # curl --connect-timeout 2 nginx.testing
.....
<title>Welcome to nginx!</title>
.....
/ #
```

第四步: 定义网络策略配置清单**myapp-allow.yaml**, 放行**testing**名称空间中来自**nginx Pod**的发往**myapp Pod**的80/TCP的访问流量, 以及**myapp Pod**发往**nginx Pod**的所有流量。另外, 允许**myapp Pod**与**kube-system**名称空间的任何**Pod**进行交互的所有流量:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: myapp-allow
  namespace: testing
spec:
  podSelector:
    matchLabels:
      app: myapp
  ingress:
    - from:
      - podSelector:
          matchLabels:
            app: nginx
        ports:
          - port: 80
      - from:
          namespaceSelector:
            matchLabels:
              ns: kube-system
  egress:
    - to:
      - podSelector:
          matchLabels:
```

```
        app: nginx
- to:
  - namespaceSelector:
      matchLabels:
        ns: kube-system
policyTypes:
- Ingress
- Egress
```

接下来首先创建清单中定义的网络策略：

```
~]$ kubectl apply -f myapp-allow.yaml
networkpolicy.networking.k8s.io "myapp-allow" configured
```

而后切换至此前在专用终端中创建的临时使用的交互式Pod，对myapp Pod发起访问请求。由下面的命令结果可知，其访问被拒绝：

```
# curl --connect-timeout 2 http://myapp.testing
curl: (28) connect() timed out!
/ #
```

myapp Pod仅允许来自nginx Pod对其80/TCP端口的访问，于是，这里进入testing名称空间中的是nginx Pod的交互式接口，使用wget命令对myapp Pod发起访问请求，如下面的命令所示：

```
~]$ kubectl exec -it nginx-b477df957-jb2nf -n testing -- /bin/sh
/ # wget http://myapp.testing -O - -q
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
/ #
```

需要注意的是，这里的nginx Pod经由Deployment控制器创建，其名称格式为两级Hash字符串，读者在执行测试操作时要转换为实际创建的标识符。如果所有测试均能通过，则表示网络策略都已经正常生效。此示例中涉及的访问控制基本上能够类比到大多数场景中策略设置的需求，这里还请读者细细揣摩其设置逻辑。

11.4 Calico网络插件

Calico是一个开源虚拟化网络方案，用于为云原生应用实现互联及策略控制。与Flannel相比，Calico的一个显著优势是对网络策略（network policy）的支持，它允许用户动态定义ACL规则控制进出容器的数据报文，实现为Pod间的通信按需施加安全策略。事实上，Calico可以整合进大多数主流的编排系统，如Kubernetes、Apache Mesos、Docker和OpenStack等。

Calico本身是一个三层的虚拟网络方案，它将每个节点都当作路由器（router），将每个节点的容器都当作是“节点路由器”的一个终端并为其分配一个IP地址，各节点路由器通过BGP（Border Gateway Protocol）学习生成路由规则，从而将不同节点上的容器连接起来。因此，Calico方案其实是一个纯三层的解决方案，通过每个节点协议栈的三层（网络层）确保容器之间的连通性，这摆脱了flannel host-gw类型的所有节点必须位于同一二层网络的限制，从而极大地扩展了网络规模和网络边界。如图11-18所示的是Calico系统示意图。

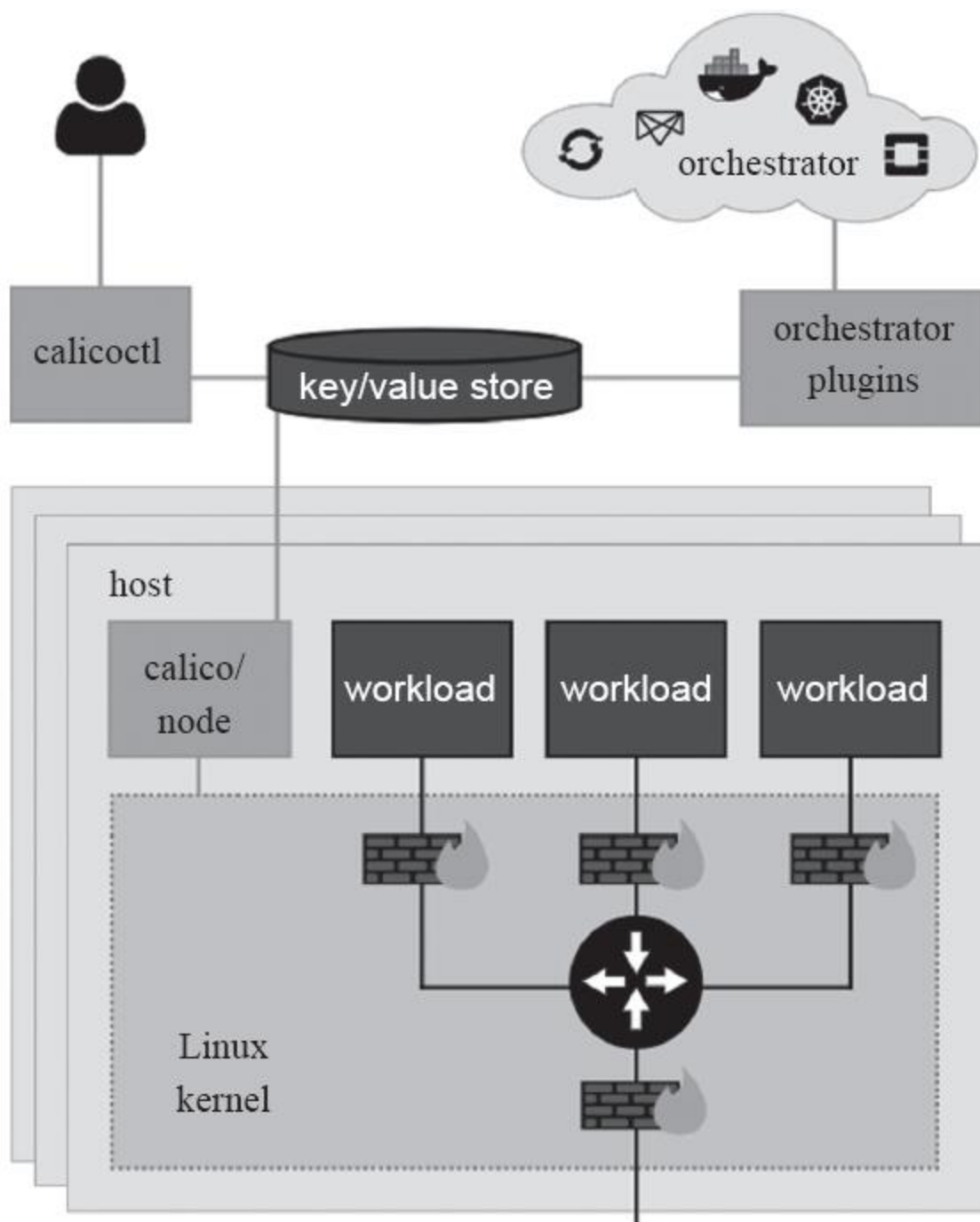


图11-18 Calico系统示意图

BGP是互联网上一个核心的去中心化自治路由协议，它通过维护IP路由表或“前缀”表来实现自治系统（AS）之间的可达性，属于矢量路由协议。不过，考虑到并非所有的网络都能支持**BGP**，以及**Calico**控制平面的设计要求物理网络必须是二层网络，以确保vRouter间均直接可达，路由不能够将物理设备当作下一跳等原因，为了支持三层网

络，Calico还推出了IP-in-IP叠加的模型，它也使用Overlay的方式来传输数据。IPIP的包头非常小，而且也是内置在内核中，因此理论上它的速度要比VxLAN快一点，但安全性更差。Calico 3.x的默认配置使用的是IPIP类型的传输方案而非BGP。

11.4.1 Calico工作特性

Calico利用Linux内核在每一个计算节点上实现了一个高效的vRouter（虚拟路由器）进行报文转发，而每个vRouter都通过BGP负责把自身所属的节点上运行的Pod资源的IP地址信息基于节点的agent程序（Felix）直接由vRouter生成路由规则向整个Calico网络内进行传播，不过，尽管小规模部署可以直接互联，但大规模网络还是建议使用BGP路由反射器（route reflector）来完成。Felix也支持在每个节点上按需生成ACL（Access Control List）从而实现安全策略，如隔离不同的租户或项目的网络通信。vRouter利用BGP通告本节点上现有的地址分配信息，每个vRouter均接入BGP路由反射器以实现控制平面扩展。

Calico承载的各Pod资源直接通过vRouter经由基础网络进行互联，它非叠加、无隧道、不使用VRF表，也不依赖于NAT，因此每个工作负载都可以直接配置使用公网IP接入互联网，当然，也可以按需使用网络策略控制它的网络连通性。

（1）经IP路由直连

Calico中，Pod收发的IP报文由所在节点的Linux内核路由表负责转发，并通过iptables规则实现其安全功能。某Pod对象发送报文时，Calico应确保节点总是作为下一跳MAC地址返回，不管工作负载本身可能配置什么路由，而发往某Pod对象的报文，其最后一个IP跃点就是Pod所在的节点，也就是说，报文的最后一程即由节点送往目标Pod对象，如图11-19所示。

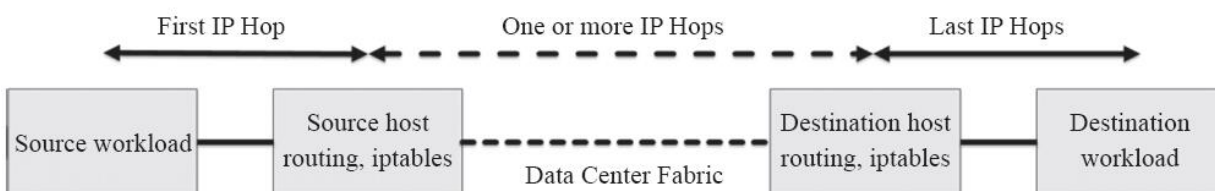


图11-19 经IP路由直连

需为某Pod对象提供连接时，系统上的专用插件（如Kubernetes的CNI）负责将需求通知给Calico Agent。收到消息后，Calico Agent会为每个工作负载添加直接路径信息到工作负载的TAP设备（如veth）。而

运行于当前节点的BGP客户端监控到此类消息后会调用路由reflector向工作于其他节点的BGP客户端进行通告。

(2) 简单、高效、易扩展

Calico未使用额外的报文封装和解封装，从而简化了网络拓扑，这也是Calico高性能、易扩展的关键因素。毕竟，小的报文减少了报文分片的可能性，而且较少的封装和解封装操作也降低了对CPU的占用。此外，较少的封装也易于实现报文分析，易于进行故障排查。

创建、移动或删除Pod对象时，相关路由信息的通告速度也是影响其扩展性的一个重要因素。Calico出色的扩展性缘于与互联网架构设计原则别无二致的方式，它们都使用了BGP作为控制平面。BGP以高效管理百万级的路由设备而闻名于世，Calico自然可以游刃有余地适配大型IDC网络规模。另外，由于Calico各工作负载使用基IP直接进行互联，因此它还支持多个跨地域的IDC之间进行协同。

(3) 较好的安全性

原则上，Calico网络允许IDC中的任何工作负载与其他任意目标进行通信，但管理员或用户却未必期望如此，在多租户IDC中隔离租户网络几乎是必然之需。于是，Calico操纵节点上的iptables规则以管控工作负载的互联许可。此种iptables规则操纵功能是节点间的警戒哨，负责阻挡任何非许可流量，并防止通过工作负载危及节点自身。

·严格的域间流量分隔：运行于某个租户虚拟网络内的应用应严禁访问其他租户的应用，这种流量分隔是由久经考验的Linux内核中的ACL子系统予以实现的。

·精细的策略规则：实现了租户间隔离的网络方案大多并没有额外实现细粒度的安全策略，而Calico通过使用Linux内建的ACL扩展来支持一众安全规则，任何可由ACL支持的功能均能通过Calico实现。

·简洁而不简单：Calico直接使用IP网络，无须任何地址转换或隧道承载的机制实现了一个简洁的“WYSIWYG”（What You See Is What You Get）网络模型，它可以清晰地标识出每个报文从哪儿来，到哪儿去。于是，管理员可因此而清晰地理解流量的来去。

11.4.2 Calico系统架构

概括来说，Calico主要由Felix、Orchestrator Plugin、etcd、BIRD和BGP Router Reflector等组件组成，其组件架构如图11-20所示。

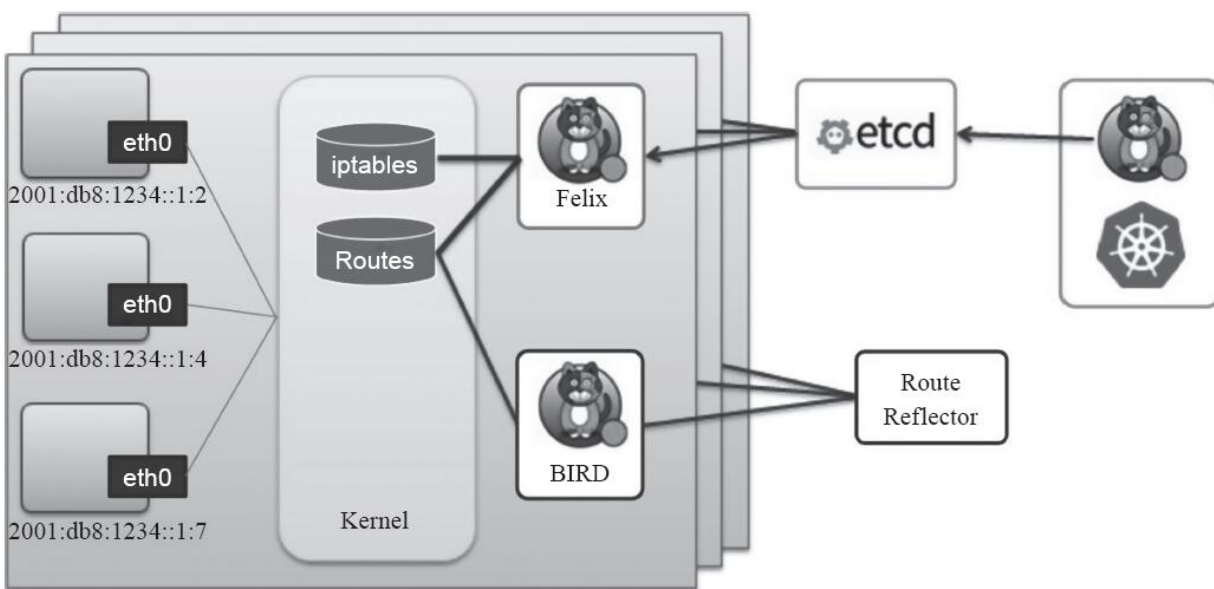


图11-20 Calico系统组件

- Felix: Calico Agent，运行于每个节点。
- Orchestrator Plugin: 编排系统（如Kubernetes、OpenStack等）以将Calico整合进系统中的插件，例如Kubernetes的CNI。
- etcd: 持久存储Calico数据的存储管理系统。
- BIRD: 用于分发路由信息的BGP客户端。
- BGP Route Reflector: BGP路由反射器，可选组件，用于较大规模的网络场景。

1.Felix

Felix运行于各节点的用于支持端点（VM或Container）构建的守护进程，它负责生成路由和ACL，以及其他任何由节点用到的信息，从

而为各端点构建连接机制。**Felix**在各编排系统中主要负责以下任务。

首先是接口管理（**Interface Management**）功能，负责为接口生成必要的信息并送往内核，以确保内核能够正确处理各端点的流量，尤其是要确保各节点能够响应目标**MAC**为当前节点上各工作负载的**MAC**地址的**ARP**请求，以及为其管理的接口打开转发功能。另外，它还要监控各接口的变动以确保规则能够得到正确的应用。

其次是路由规划（**Route Programming**）功能，其负责为当前节点运行的各端点在内核**FIB**（**Forwarding Information Base**）中生成路由信息，以保证到达当前节点的报文可正确转发给端点。

再次是**ACL**规划（**ACL Programming**）功能，负责在Linux内核中生成**ACL**，用于实现仅放行端点间的合法流量，并确保流量不能绕过**Calico**的安全措施。

最后是状态报告（**State Reporting**）功能，负责提供网络健康状态的相关数据，尤其是报告由其管理的节点上的错误和问题。这些报告数据会存储于**etcd**，供其他组件或网络管理员使用。

2.编排系统插件

编排系统插件（**Orchestrator Plugin**）依赖于编排系统自身的实现，故此并不存在一个固定的插件以代表此组件。编排系统插件的主要功能是将**Calico**整合进系统中，并让管理员和用户能够使用**Calico**的网络功能。它主要负责完成**API**的转换和反馈输出。

编排系统通常有其自身的网络管理**API**，网络插件需要负责将对这些**API**的调用转为**Calico**的数据模型并存储于**Calico**的存储系统中。如果有必要，网络插件还要将**Calico**系统的信息反馈给编排系统，如**Felix**的存活状态，网络发生错误时设定相应的端点为故障等。

3.etcd存储系统

Calico使用**etcd**完成组件间的通信，并以之作为一个持久数据存储系统。根据编排系统的不同，**etcd**所扮演角色的重要性也因之而异，但它贯穿了整个**Calico**部署全程，并被分为两类主机：核心集群和代理（**proxy**）。在每个运行着**Felix**或编排系统插件的主机上都应该运行一

个etcd代理以降低etcd集群和集群边缘节点的压力。此模式中，每个运行着插件的节点都会运行着etcd集群的一个成员节点。

etcd是一个分布式、强一致、具有容错功能的存储系统，这一点有助于将Calico网络实现为一个状态确切的系统：要么正常，要么发生故障。另外，分布式存储易于通过扩展应对访问压力的提升，而避免成为系统瓶颈。另外，etcd也是Calico各组件的通信总线，可用于确保让非etcd组件在键空间（keyspace）中监控某些特定的键，以确保它们能够看到所做的任何更改，从而使它们能够及时地响应这些更改。

4.BGP客户端（BIRD）

Calico要求在每个运行着Felix的节点上同时还要运行一个BGP客户端，负责将Felix生成的路由信息载入内核并通告到整个IDC。在Calico语境中，此组件是通用的BIRD，因此任何BGP客户端（如GoBGP等）都可以从内核中提取路由并对其分发对于它们来说都适合的角色。

BGP客户端的核心功能就是路由分发，在Felix插入路由信息至内核FIB中时，BGP客户端会捕获这些信息并将其分发至其他节点，从而确保了流量的高效路由。

5.BGP路由反射器（BIRD）

在大规模的部署场景中，简易版的BGP客户端易于成为性能瓶颈，因为它要求每个BGP客户端都必须连接至其同一网络中的其他所有BGP客户端以传递路由信息，一个有着N个节点的部署环境中，其存在网络连接的数量为N的二次方，随着N值的逐渐增大，其连接复杂度会急剧上升。因而在较大规模的部署场景中，Calico应该选择部署一个BGP路由反射器，它是由BGP客户端连接的中心点，BGP的点ToPoint通信也就因此转化为与中心点的单路通信模型，如图11-18所示。出于冗余之需，生产实践中应该部署多个BGP路由反射器。对于Calico来说，BGP客户端程序除了作为客户端使用之外，还可以配置成路由反射器。

11.4.3 Calico部署要点

安装Calico需要事先有运行中的Kubernetes 1.1及以上版本的集群，如果要用到网络策略，则Kubernetes版本要1.3.0及以上才可以。另外，还需要一个可由Kubernetes集群各节点访问到的etcd集群。虽然可以让Calico和Kubernetes共同使用同一个etcd集群，然则有些场景中推荐为etcd使用专用集群，如需要获得较好的性能时。

与Kubernetes集群进行整合时，Calico需要提供三个组件，具体如下。

- calico/node: Calico于Kubernetes集群中为每个节点上运行的容器提供Felix Agent和BGP客户端。

- cni-plugin: CNI网络插件，用于整合Calico和kubelet，发现Pod资源，并将其添加进Calico网络，因此，每个运行kubelet的主机都需要配置。

- calico/kube-controllers: Calico网络策略控制器。

Calico的安装有两种方式，一是配置其独立运行于Kubernetes集群之外，但calico/kube-controllers依然需要以Pod资源运行于集群之上。另一种是以插件方式配置Calico完全托管运行于Kubernetes集群之上，不过此种方式要求Kubernetes版本至少在1.4.0以上。

另外，Calico以Kubernetes插件方式部署的实现方式有两种，一是标准托管式部署，即Calico使用专用的etcd存储管理数据持久化及组件间的通信。另一个是以Kubernetes API Server为Datastore，调用Kubernetes的API完成所需的操作。自3.0版本起，Calico官方推荐使用第二种方式，而此前的版本中此种功能尚且不够完善，所推荐的则是第一种部署方式。

再者，Calico既可以单独为Kubernetes系统提供网络服务及网络策略，也可以与flannel整合在一起由flannel实现网络服务而Calico仅提供网络策略。事实上，还有一个本身即是将flannel或Calico合二为一的解决方案Canal，不过，这已经是另一个独立的项目。

需要注意的是，Calico分配的地址池与Kubernetes集群的--pod-network-cidr的值应该保持一致，默认情况下，Calico的配置清单中使用192.168.0.0/16作为Pod网络。不过，如果用户计划将Calico和flannel协同进行部署，则可以在此前已有flannel插件的基础上直接添加Calico。11.4.4节将在讲解独立部署Calico的同时提供网络服务及网络策略，协同flannel的部署方式请读者参考官方文件中的相关介绍进行操作。

11.4.4 部署Calico提供网络服务和网络策略

为了便于简化部署环境及操作复杂度，本节的操作建立在一个刚由kubeadm部署完成的新的Kubernetes集群之上，部署时为--pod-network-cidr选项指定了使用192.168.0.0/16网络以适配Calico的默认网络配置，它还没有添加过任何网络插件。Calico的部署方式极其灵活，这里难以尽述其所有的实现方式，仅以典型应用场景对其加以说明。

Calico 3目前仅支持Kubernetes 1.8及其之后的版本，并且它要求使用一个能够被各组件访问到的键值存储系统，在Kubernetes环境中，可用的选择有etcd v3或Kubernetes API数据存储。本部署示例会将Kubernetes API作为Calico的数据存储取代etcd，这也是在本章写作时下最新的稳定版本Calico 3版本中推荐的配置。另外，Calico 3目前对kube-proxy ipvs模式的支持尚且处于试用级别，因此不建议读者在生产环境中使用。



注意 不同版本的Calico的部署方式可能不尽相同，读者需要根据操作时的具体情况参考官方文档来确定具体的部署方案及部署步骤。

在启用了RBAC的Kubernetes集群部署Calico时，需要先创建必要的ClusterRole和ClusterRoleBinding资源。Calico官方给出了标准定义的配置清单，部署时，直接应用在线清单即可，如下面的命令所示：

```
~]$ kubectl apply -f https://docs.projectcalico.org/v3.2/getting-started/kubernetes/installation/hosted/rbac-kdd.yaml
```

应用完成后，它会创建名为calico-node的clusterrole和clusterrolebinding，为相关的Pod资源calico-node的ServiceAccount授予必要的资源管理权限。Kubernetes集群规模小于50节点时，可以类似如下命令部署Calico的各相关组件，它们会创建多个标准的Kubernetes资源及数个自定义的资源：

```
~]$ kubectl apply -f https://docs.projectcalico.org/v3.2/getting-  
started/kubernetes/  
installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml
```

可通过如下命令查看calico-node相关的Pod资源于各节点中的部署状态。待所有Pod资源均处于“Running”状态后，即可正常使用其相关的功能：

```
~]$ kubectl get pods -l k8s-app=calico-node -o wide -n kube-system
```

工作于IPIP模式的Calico会在每个节点上创建一个tunl0接口（TUN类型虚拟设备）用于封装三层隧道报文。节点上创建的每一个Pod资源，都会由Calico自动创建一对虚拟以太网接口（TAP类型的虚拟设备），其中一个附加于Pod的网络名称空间，另一个（名称以cali为前缀后跟随机字串）留置在节点的根网络名称空间，并经由tunl0封装或解封三层隧道报文。Calico IPIP模式如图11-21所示。

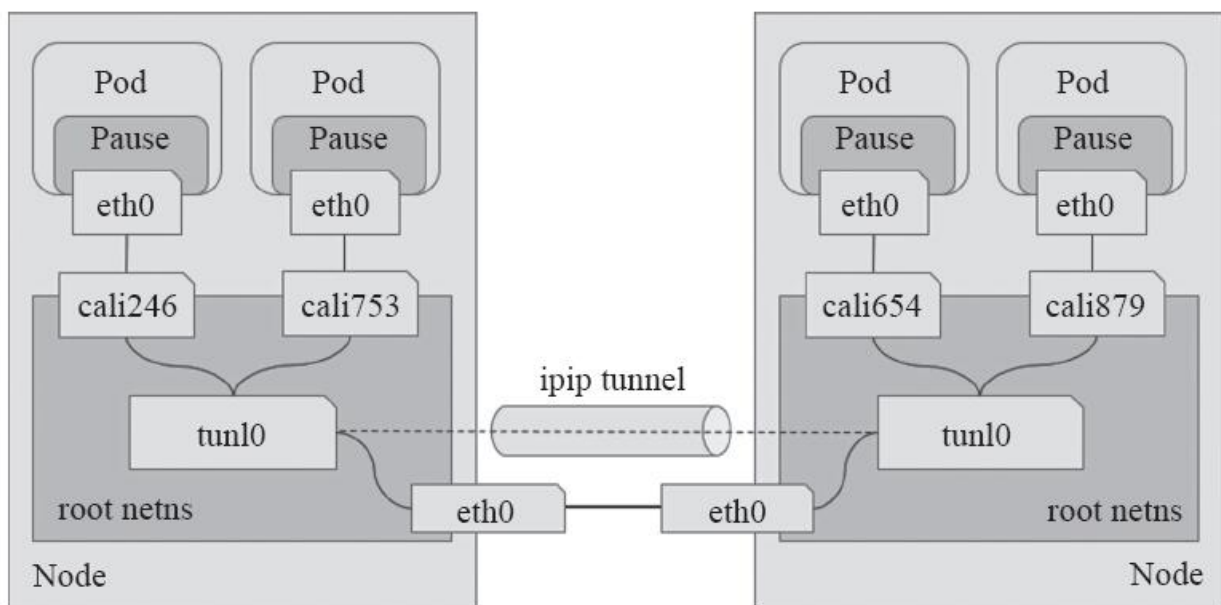


图11-21 Calico IPIP

部署完成后，Calico会在每个节点上生成到达Kubernetes集群中每个节点上的Pod子网的路由信息，下面所示的路由信息是部署示例中node01主机上生成的路由条目，它们由各节点上的BIRD以点对点的方

```
192.168.0.0/24 via 172.16.0.70 dev tunl0 proto bird onlink
blackhole 192.168.1.0/24 proto bird
192.168.2.0/24 via 172.16.0.67 dev tunl0 proto bird onlink
192.168.3.0/24 via 172.16.0.68 dev tunl0 proto bird onlink
```

在每个节点上创建Pod资源时，由Calico CNI插件为其生成TAP设备并分配地址后，也会在节点的网络名称空间中生成一个新的路由条目，类似如下所示，它指明了发往本节点上某特定Pod IP的报文应该经由的TAP接口：

```
192.168.1.3 dev cali8bb05ff8b64 scope link
```

在集群中部署一些Pod资源即可完成集群网络连接测试。假设此时在node01上存在一个IP地址为192.168.1.3的Pod A，以及在node02上存在一个IP地址为192.168.2.4的Pod B，通过Pod A的交互式接口对Pod B发起ping请求，在node01的物理接口上以root用户的身份抓取通信报文，命令及截取的一次通信的往返结果示例如下：

```
~]# tcpdump -i ens33 -nn ip host 172.16.0.66 and host 172.16.0.67
10:58:03.553589 IP 172.16.0.66 > 172.16.0.67: IP 192.168.1.3 > 192.168.2.4: ICMP
  echo request, id 4096, seq 33, length 64 (ipip-proto-4)
10:58:03.553883 IP 172.16.0.67 > 172.16.0.66: IP 192.168.2.4 > 192.168.1.3: ICMP
  echo reply, id 4096, seq 33, length 64 (ipip-proto-4)
```

命令结果显示，Pod间的通信经由IPIP的三层隧道转发，外层IP首部中的IP地址为通信双方的节点IP（172.16.0.66和172.16.0.67），内层IP首部中的IP地址为通信双方的Pod IP（192.168.1.3和192.168.2.4）。相比较VxLAN的二层隧道来说，IPIP隧道的开销较小，但其安全性也更差一些。

需要提醒读者注意的是，tunl0接口的MTU默认为1440，这种设置主要是为适配Google的GCE环境，在非GCE的物理环境中，其最佳值为1480。因此，对于非GCE环境的部署，建议将配置清单calico.yaml下载至本地修改后，再将其应用到集群中。要修改的内容是DaemonSet资源calico-node的Pod模板，将容器calico-node的环境变量“FELIX_INPUTMTU”的值修改为1480即可，类似如下所示。图11-22给出了Calico部署于不同网络环境时不同部署方式适用的MTU大小。

Network MTU	Calico MTU	Calico MTU with IP-in-IP	Calico MTU with VxLAN (IPv4)
1500	1500	1480	1450
9000	9000	8980	8950
1460 (GCE)	1460	1440	1410
9001 (AWS Jumbo)	9001	8981	8951

图11-22 Calico于各网络环境上适用的MTU

对于50个节点以上规模的集群来说，所有Calico节点均基于Kubernetes API存取数据会为API Server带来不小的通信压力，这就应该使用calico-typha进程将所有Calico的通信集中起来与API Server进行统一交互。calico-typha以Pod资源的形式托管运行于Kubernetes系统之上，启用的方法为下载前面步骤中用到的Calico的部署清单文件至本地，修改其calico-typha的Pod资源副本数量为所期望的值并重新应用配置清单即可：

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: calico-typha
  ...
spec:
  ...
  replicas: <number of replicas>
```

每个calico-typha Pod资源可承载100到200个Calico节点的连接请求，最多不要超过200个。另外，整个集群中的calico-typha的Pod资源总数尽量不要超过20个。

11.4.5 客户端工具calicoctl

Calico的二进制程序文件calicoctl可直接操作Calico存储来查看、修改或配置Calico系统特性，它可以运行为Kubernetes系统之上的Pod资源，也可直接以裸二进制文件部署于某管理主机之上，例如，kubectl所在的某主机以root用户的身份下载calicoctl文件并直接保存于/usr/bin/目录中：

```
~]# wget
https://github.com/projectcalico/calicoctl/releases/download/v3.1.1/calicoctl \
-O /usr/bin/calicoctl
~]# chmod +x /usr/bin/calicoctl
```

calicoctl通过读写Calico的数据存储系统（datastore）进行查看或进行各类管理操作，通常，它需要提供认证信息经由相应的数据存储完成认证。使用Kubernetes API数据存储时，需要使用类似kubectl的认证信息完成认证。它可以通过环境变量声明的DATASTORE_TYPE和KUBECONFIG接入Kubernetes集群，例如以如下命令格式运行calicoctl：

```
~]$ DATASTORE_TYPE=kubernetes KUBECONFIG=~/.kube/config calicoctl get nodes
```

也可以直接将认证信息等保存于配置文件中，calicoctl默认加载/etc/calico/calicoctl.cfg配置文件读取配置信息。配置文件为yaml格式，语法极其类似于Kubernetes的资源配置清单：

```
apiVersion: projectcalico.org/v3
kind: CalicoAPIConfig
metadata:
spec:
  datastoreType: "kubernetes"
  kubeconfig: "/PATH/TO/.kube/config"
```

将上面示例配置中的/PATH/TO路径修改为相应的用户家目录即可，如/home/ik8s/。当然，也可以是用户自定义的其他kubeconfig配置文件的存放路径。

`calicoctl`的通用语法格式为“`calicoctl[options]<command> [<args>...]`”，它有着多个子命令，用于增删改查相应的配置及状态信息等。例如，可使用如下命令查看当前Calico部署的相关节点状态信息：

```
~]$calicoctl node status
```

默认情况下，Calico的BGP网络工作于点对点的网格（**node-to-node mesh**）模型，它仅适用于较小规模的集群环境。中级集群环境应该使用全局对等BGP模型（**Global BGP peers**），以在同一二层网络中使用一个或一组BGP反射器构建BGP网络环境。而大型集群环境需要使用每节点对等BGP模型（**Per-node BGP peers**），即分布式BGP反射器模型，一个典型的用法是将每个节点都配置为自带BGP反射器接入机架顶部交换机上的路由反射器。

另外，读者也可以通过如下命令了解Calico当前IP地址池的相关设定，包括其地址范围，是否启用了IPIP模型等：

```
~]$ calicoctl get ipPool -o yaml
apiVersion: projectcalico.org/v3
items:
- apiVersion: projectcalico.org/v3
  kind: IPPool
  metadata:
  .....
```

事实上，仅在那些不支持用户自定义BGP配置的网络中才需要使用IPIP的隧道通信类型。如果读者有一个自主可控的网络环境且部署规模较大时，可以考虑启用BGP的通信类型降低网络开销以提升传输性能，并且应该部署BGP反射器来提高路由学习效率。具体的实现方式请参考Calico站点上<https://docs.projectcalico.org> 给出的文档。

11.5 本章小结

本章详细描述了Kubernetes的网络模型、CNI插件体系、主流的CNI插件flannel和Calico，并重点介绍了flannel和Calico的特性和应用方式，具体如下。

- Kubernetes的网络模型中包含容器间通信、Pod间通信、Service与Pod间的通信，以及集群外部流量与Pod间的通信四种通信需求。

- Kubernetes网络模型的实现通过CNI接口由外部网络插件来实现，如flannel、Calico和Canal等。

- flannel支持host-gw、VxLAN和UDP等后端，默认为VxLAN。

- 网络策略能够给Pod间提供通信隔离机制，它支持Ingress和Egress两种类型的规则。

- Calico是另一个流行的网络插件，它提供了高性能的适用于大规模网络模型的虚拟网络。

第12章 Pod资源调度

API Server接受客户端提交Pod对象创建请求后的操作过程中，有一个重要的步骤是由调度器程序（**kube-scheduler**）从当前集群中选择一个可用的最佳节点来接收并运行它，通常是默认的调度器（**default-scheduler**）负责执行此类任务。对于每个待创建的Pod对象来说，调度过程通常分为三个阶段—预选、优选和选定三个步骤，以筛选执行任务的最佳节点。本章将重点描述这三个步骤的工作过程。

12.1 Kubernetes调度器概述

Kubernetes系统的核心任务在于创建客户端请求创建的Pod对象并确保其以期望的状态运行。创建Pod对象时，调度器（scheduler）负责为每一个未经调度的Pod资源、基于一系列的规则集从集群中挑选一个合适的节点来运行它，因此它也可以称作Pod调度器。调度过程中，调度器不会修改Pod资源，而是从中读取数据，并根据配置的策略挑选出最适合的节点，而后通过API调用将Pod绑定至挑选出的节点之上以完成调度过程，如图12-1所示。

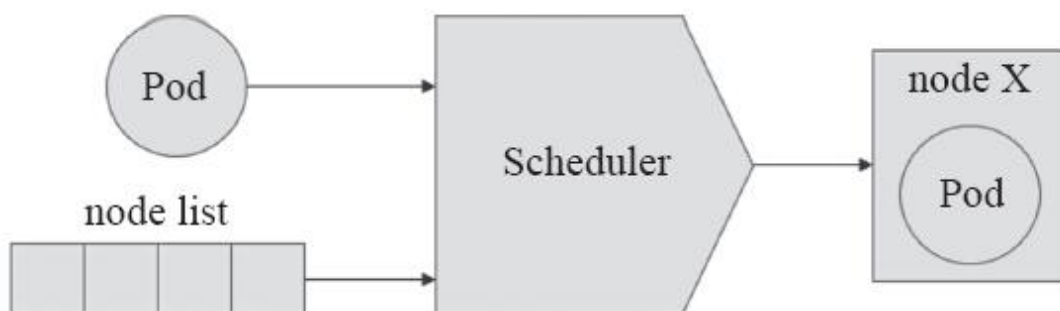


图12-1 Kubernetes调度器

Kubernetes内建了适合绝大多数场景中Pod资源调度需求的默认调度器，它支持同时使用算法基于原生及可定制的工具来选出集群中最适合运行当前Pod资源的一个节点，其核心目标是基于资源可用性将各Pod资源公平地分布于集群节点之上。目前，平台提供的默认调度器也称为“通用调度器”，它通过三个步骤完成调度操作：节点预选（Predicate）、节点优先级排序（Priority）及节点择优（Select），如图12-2所示。

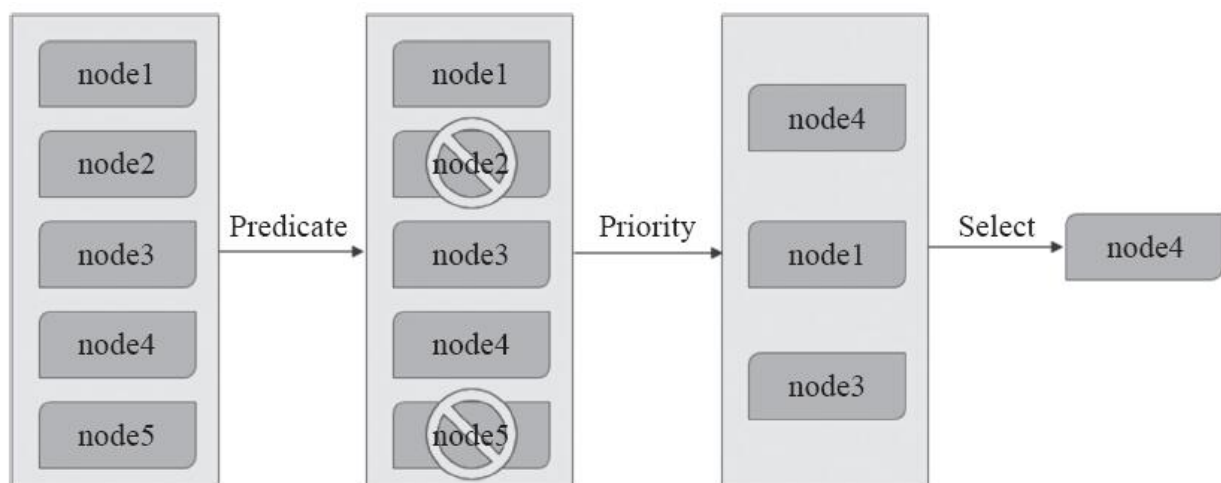


图12-2 节点预选、优选及选定示意图

1) 节点预选：基于一系列预选规则（如**PodFitsResources**和**MatchNode-Selector**等）对每个节点进行检查，将那些不符合条件的节点过滤掉从而完成节点预选。

2) 节点优选：对预选出的节点进行优先级排序，以便选出最适合运行**Pod**对象的节点。

3) 从优先级排序结果中挑出优先级最高的节点运行**Pod**对象，当此类节点多于一个时，则从中随机选择一个。

偶尔，有些特殊的**Pod**资源需要运行在特定的节点之上，或者说对某类节点有着特殊偏好（如那些有着**SSD**、**GPU**等特殊硬件的节点），以便更好地匹配容器应用的运行需求。另外，有的**Pod**资源与其他**Pod**资源存在着特定的关联性，它们运行于同一节点以便能够实现更高效的协同效果等。此种场景可通过组合节点标签，以及**Pod**标签或标签选择器等来激活特定的预选策略以完成高级调度，如**MatchInterPodAffinity**、**MatchNodeSelector**和**PodToleratesNodeTaints**等预选策略，它们用于为用户提供自定义**Pod**亲和性或反亲和性、节点亲和性以及基于污点及容忍度的调度机制。

不过，未激活特定的预选策略时，**Pod**资源对节点便没有特殊偏好，相关的预选策略无法在节点预选过程中真正发挥作用。

12.1.1 常用的预选策略

简单来说，预选策略就是节点过滤器，例如节点标签必须能够匹配到Pod资源的标签选择器（由MatchNodeSelector实现的规则），以及Pod容器的资源请求量不能大于节点上剩余的可分配资源（由PodFitsResources实现的规则）等。执行预选操作时，调度器将对每个节点基于配置使用的预选策略以特定次序逐一筛查，并根据一票否决制进行节点淘汰。若预选后不存在任何一个满足条件的节点，则Pod被置于Pending状态，直到至少有一个节点可用为止。目前，Kubernetes 1.10支持的预选策略如图12-3所示。

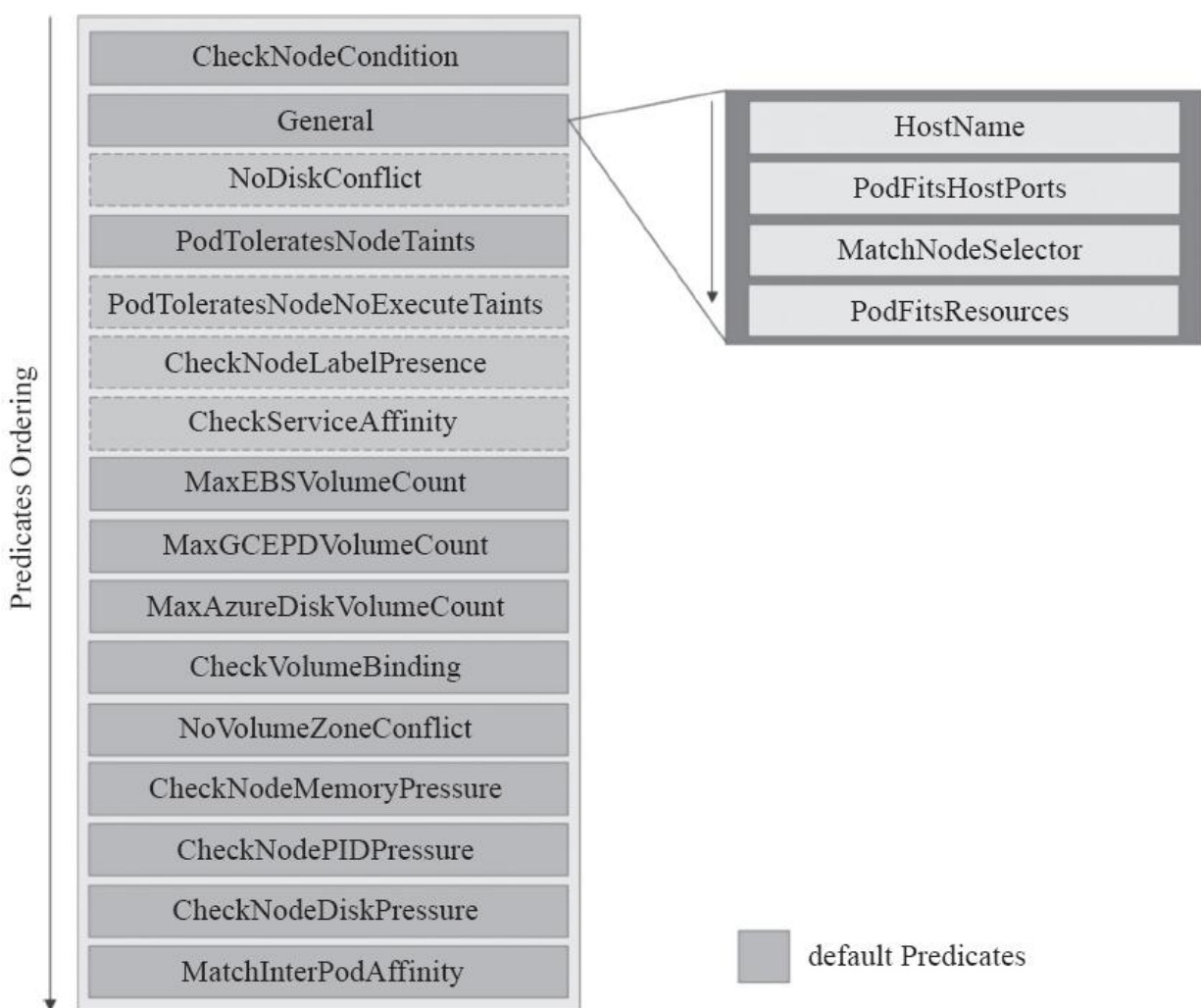


图12-3 Kubernetes的预选策略

1) **CheckNodeCondition**: 检查是否可以在节点报告磁盘、网络不可用或未准备好的情况下将Pod对象调度于其上。

2) **HostName**: 若Pod对象拥有spec.hostname属性，则检查节点名称字符串与此属性值是否匹配。

3) **PodFitsHostPorts**: 若Pod容器定义了ports.hostPort属性，则检查其值指定的端口是否已被节点上的其他容器或服务占用。在Kubernetes 1.0版本之前此预选策略名称为PodFitsPorts。

4) **MatchNodeSelector**: 若Pod对象定义了spec.nodeSelector属性，则检查节点标签是否能匹配此属性值。

5) **NoDiskConflict**: 检查Pod对象请求的存储卷在此节点是否可用，若不存在冲突则通过检查。

6) **PodFitsResources**: 检查节点是否有足够的资源（如CPU、内存和GPU等）满足Pod对象的运行需求。节点声明其资源可用容量，而Pod则定义其资源需求，于是，调度器会判断节点是否有足够的可用资源运行Pod对象，若无法满足则返回失败原因（例如，CPU或内存资源不足等）。调度器评判资源消耗的标准是节点已分配的资源量（各容器的requests值之和），而非其上各Pod对象已用的资源量。注意，那些在注解中标记为关键性（critical）的Pod资源不受此预选策略控制。

7) **PodToleratesNodeTaints**: 若Pod对象定义了spec.tolerations属性，则检查其值是否能够接纳节点定义的污点（taints），不过，它仅关注具有NoSchedule和NoExecute两个效用标识的污点。

8) **PodToleratesNodeNoExecuteTaints**: 若Pod对象定义了spec.tolerations属性，则检查其值是否能够接纳节点定义的NoExecute类型的污点。

9) **CheckNodeLabelPresence**: 仅检查节点上指定的所有标签的存在性，要检查的标签以及其可否存在取决于用户的定义。当集群中部署的节点以regions/zones/racks的拓扑方式放置且基于此类标签对其进行位置标识时，预选策略可以根据此类标识将Pod资源调度至此类节点之上。

10) **CheckServiceAffinity**: 根据当前Pod对象所属的Service已有的其他Pod对象所运行的节点进行调度，其目的在于将相同Service的Pod对象放在同一个或同一类节点上以提高效率。此预选策略试图将那些在其节点选择器中带有特定标签的Pod资源调度至拥有同样标签的节点之上，具体的标签则取决于用户的定义。例如，若某Service的第一个Pod有一个节点选择器rack，而且它被调度到了标签为region=rack的节点，则属于此Service的所有其他后续的Pod对象都将被调度至具有相同标签（region=rack）的节点上。若该Pod未在其节点选择器中指定标签，则第一个Pod将根据可用性放置在任一节点之上，而后该Service的所有后续Pod对象都将调度至与该节点拥有相同标签值的节点上。

11) **MaxEBSVolumeCount**: 检查节点上已挂载的EBS存储卷数量是否超过了设置的最大值，默认值为39。

12) **MaxGCEPDVolumeCount**: 检查节点上已挂载的GCE PD存储卷数量是否超过了设置的最大值，默认值为16。

13) **MaxAzureDiskVolumeCount**: 检查节点上已挂载的Azure Disk存储卷数量是否超过了设置的最大值，默认值为16。

14) **CheckVolumeBinding**: 检查节点上已绑定和未绑定的PVC是否能够满足Pod对象的存储卷需求，对于已绑定的PVC，此预选策略将检查给定的节点是否能够兼容相应的PV，而对于未绑定的PVC，预选策略将搜索那些可满足PVC申请的可用PV，并确保它可与给定的节点兼容。

15) **NoVolumeZoneConflict**: 在给定了区域（zone）限制的前提下，检查在此节点上部署Pod对象是否存在存储卷冲突。某些存储卷存在区域调度约束，于是，此类存储卷的区域标签（zone-labels）必须与节点上的区域标签完全匹配方可满足绑定条件。

16) **CheckNodeMemoryPressure**: 若给定的节点已经报告了存在内存资源压力过大的状态，则检查当前Pod对象是否可调度至此节点之上。目前，最低优先级的BestEffort QoS类型的Pod资源也不可调度至此类节点之上，因为它们随时可能因OOM而终止。

17) **CheckNodePIDPressure**: 若给定的节点已经报告了存在PID资源压力过大的状态，则检查当前Pod对象是否可调度至此节点之上。

18) **CheckNodeDiskPressure**: 若给定的节点已经报告了存在磁盘资源压力过大的状态，则检查当前Pod对象是否可调度至此节点之上。

19) **MatchInterPodAffinity**: 检查给定节点是否能够满足Pod对象的亲和性或反亲和性条件，以用于实现Pod亲和性调度或反亲和性调度。

如上给定的各预选策略中，**CheckNodeLabelPresence**和**CheckServiceAffinity**可以接受特定的配置参数以便在预选过程中融合用户自定义的调度逻辑，这类策略也可称为可配置策略，而余下那些不可接受配置参数的策略也统一称为静态策略。另外，**NoDiskConflict**、**PodToleratesNodeNoExecuteTaints**、**CheckNodeLabelPresence**和**CheckServiceAffinity**没有包含在默认的预选策略中。

12.1.2 常用的优选函数

预选策略筛选并生成一个节点列表后即进入第二阶段的优选过程。在这个过程中，调度器向每个通过预选的节点传递一系列的优选函数（如BalancedResourceAllocation和TaintTolerationPriority等）来计算其优先级分值，优先级分值介于0到10之间，其中0表示不适用，10表示最适合托管该Pod对象。Kubernetes支持的优选函数如图12-4所示。

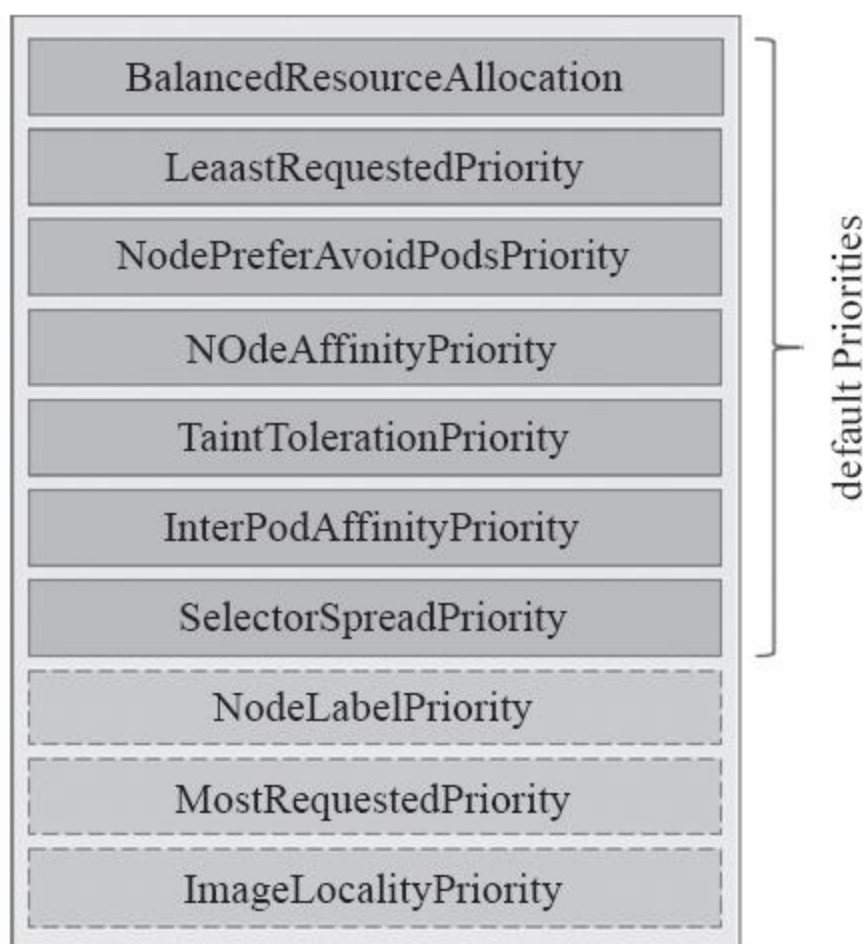


图12-4 Kubernetes支持的优选函数

另外，调度器还支持为每个优选函数指定一个简单的由正数值表示的权重，进行节点优先级分值的计算时，它首先将每个优选函数的计算得分乘以其权重（大多数优先级的默认权重为1），然后将所有优

选函数的得分相加从而得出节点的最终优先级分值。权重属性赋予了管理员定义优选函数倾向性的能力。下面是每个节点的最终优先级得分的计算公式：

$$\text{finalScoreNode} = (\text{weight1} * \text{priorityFunc1}) + (\text{weight2} * \text{priorityFunc2}) + \dots$$

下面是各优选函数的相关说明。

1) **LeastRequestedPriority**: 由节点空闲资源与节点总容量的比值计算而来，即由CPU或内存资源的总容量减去节点上已有Pod对象需求的容量总和，再减去当前要创建的Pod对象的需求容量得到的结果除以总容量。CPU和内存具有相同的权重，资源空闲比例越高的节点得分就越高，其计算公式为： $(\text{cpu}((\text{capacity} - \text{sum}(\text{requested})) * 10 / \text{capacity}) + \text{memory}((\text{capacity} - \text{sum}(\text{requested})) * 10 / \text{capacity})) / 2$ 。

2) **BalancedResourceAllocation**: 以CPU和内存资源占用率的相近程度作为评估标准，二者越接近的节点权重越高。该优先级函数不能单独使用，它需要与**LeastRequestedPriority**组合使用来平衡优化节点资源的使用状况，以选择那些在部署当前Pod资源后系统资源更为均衡的节点。

3) **NodePreferAvoidPodsPriority**: 此优先级函数权限默认为10000，它将根据节点是否设置了注解信息“[scheduler.alpha.kubernetes.io/preferAvoidPods](https://kubernetes.io/docs/scheduler.alpha.kubernetes.io/preferAvoidPods)”来计算其优先级。计算方式是：给定的节点无此注解信息时，其得分为10乘以权重10000；存在此注解信息时，对于那些由ReplicationController或ReplicaSet控制器管控的Pod对象的得分为0，其他Pod对象会被忽略（得最高分）。

4) **NodeAffinityPriority**: 基于节点亲和性调度偏好进行优先级评估，它将根据Pod资源中的nodeSelector对给定节点进行匹配度检查，成功匹配到的条目越多则节点得分越高。不过，其评估过程使用首选而非强制型的“PreferredDuringSchedulingIgnoredDuringExecution”标签选择器。

5) **TaintTolerationPriority**: 基于Pod资源对节点的污点容忍调度偏好进行其优先级的评估，它将Pod对象的tolerations列表与节点的污点

进行匹配度检查，成功匹配的条目越多，则节点得分越低。

6) **SelectorSpreadPriority**: 首先查找与当前Pod对象匹配的Service、ReplicationController、ReplicaSet (RS) 和StatefulSet，而后查找与这些选择器匹配的现存Pod对象及其所在的节点，则运行此类Pod对象越少的节点得分将越高。简单来说，如其名称所示，此优选函数会尽量将同一标签选择器匹配到的Pod资源分散到不同的节点上运行。

7) **InterPodAffinityPriority**: 遍历Pod对象的亲和性条目，并将那些能够匹配到给定节点的条目的权重相加，结果值越大的节点得分越高。

8) **MostRequestedPriority**: 与优选函数LeastRequestedPriority的评估节点得分的方法相似，不同的是，资源占用比例越大的节点，其得分越高。

9) **NodeLabelPriority**: 根据节点是否拥有特定的标签来评估其得分，而无论其值为何。需要其存在时，拥有相应标签的节点将获得优先级，否则，不具有相应标签的节点将获得优先级。

10) **ImageLocalityPriority**: 基于给定节点上拥有的运行当前Pod对象中的容器所依赖到的镜像文件来计算节点得分，不具有Pod依赖到的任何镜像文件的节点其得分为0，而拥有相应镜像文件的各节点中，所拥有的被依赖到的镜像文件其体积之和越大则节点得分越高。

Kubernetes的默认调度器以预选、优选、选定机制完成将每个新的Pod资源绑定至为其选出的目标节点上，不过，它只是Pod对象的默认调度器，使用中，用户还可以自定义调度器插件，并在定义Pod资源配置清单时通过spec.schedulerName指定即可使用。

12.2 节点亲和调度

节点亲和性是调度程序用来确定Pod对象调度位置的一组规则，这些规则基于节点上的自定义标签和Pod对象上指定的标签选择器进行定义。节点亲和性允许Pod对象定义针对一组可以调度于其上的节点的亲和性或反亲和性，不过，它无法具体到某个特定的节点。例如，将Pod调度至有着特殊CPU的节点或一个可用区域内的节点之上。

定义节点亲和性规则时有两种类型的节点亲和性规则：硬亲和性（**required**）和软亲和性（**preferred**）。硬亲和性实现的是强制性规则，它是Pod调度时必须满足的规则，而在不存在满足规则的节点时，Pod对象会被置为**Pending**状态。而软亲和性规则实现的是一种柔性调度限制，它倾向于将Pod对象运行于某类特定的节点之上，而调度器也将尽量满足此需求，但在无法满足调度需求时它将退而求其次地选择一个不匹配规则的节点。

定义节点亲和规则的关键点有两个，一是为节点配置合乎需求的标签，另一个是为Pod对象定义合理的标签选择器，从而能够基于标签选择出符合期望的目标节点。不过，如**preferredDuringSchedulingIgnoredDuringExecution**和**requiredDuringSchedulingIgnoredDuringExecution**名字中的后半段字符串**IgnoredDuringExecution**隐含的意义所指，在Pod资源基于节点亲和性规则调度至某节点之后，节点标签发生了改变而不再符合此节点亲和性规则时，调度器不会将Pod对象从此节点上移出，因为，它仅对新建的Pod对象生效。节点亲和性模型如图12-5所示。

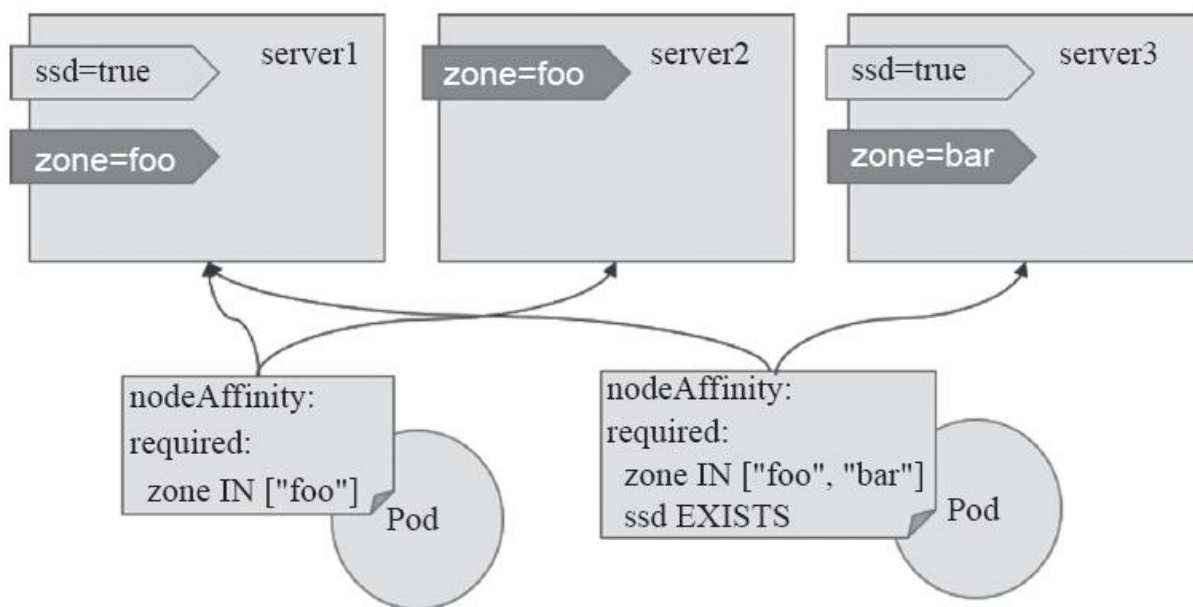


图12-5 节点亲和性

12.2.1 节点硬亲和性

为Pod对象使用nodeSelector属性可以基于节点标签匹配的方式将Pod对象强制调度至某一类特定的节点之上，这一点在第4章中曾有介绍，不过它仅能基于简单的等值关系定义标签选择器，而nodeAffinity中支持使用matchExpressions属性构建更为复杂的标签选择机制。例如，下面的配置清单示例（required-nodeAffinity-pod.yaml）中定义的Pod对象，其使用节点硬亲和规则定义可将当前Pod对象调度至拥有zone标签且其值为foo的节点之上：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-required-nodeaffinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - {key: zone, operator: In, values: ["foo"]}
  containers:
    - name: myapp
  image: ikubernetes/myapp:v1
```

将上面配置清单中定义的资源创建于集群之中，由其状态信息可知它处于Pending阶段，这是由于强制型的节点亲和限制场景中不存在能够满足匹配条件的节点所致：

```
~]$ kubectl apply -f required-nodeAffinity-pod.yaml
pod "with-required-nodeaffinity" created
~]$ kubectl get pods with-required-nodeaffinity
```

NAME	READY	STATUS	RESTARTS	AGE
with-required-nodeaffinity	0/1	Pending	0	53s

“kubectl describe”命令显示的资源详细信息Events字段中也给出了具体的原因“0/4nodes are available: 4node (s) didn't match node selector”，命令及结果如下所示：

```
~]$ kubectl describe pods with-required-nodeaffinity
.....
```

```
Events:
  Type            Reason            Age           From           Message
  ----            -
Warning FailedScheduling 6s (x6 over 21s) default-scheduler 0/4 nodes are
available: 4 node(s) didn't match node selector.
```

接下来按图12-5中的规划为各节点设置节点标签，这也是设置节点亲和性的前提之一：

```
~]$ kubectl label node node02.ilinux.io zone=foo
node "node02.ilinux.io" labeled
~]$ kubectl label node node02.ilinux.io zone=foo
node "node02.ilinux.io" labeled
~]$ kubectl label node node03.ilinux.io zone=bar
node "node03.ilinux.io" labeled
~]$ kubectl label node node01.ilinux.io ssd=true
node "node01.ilinux.io" labeled
~]$ kubectl label node node03.ilinux.io ssd=true
node "node03.ilinux.io" labeled
```

设置完成后，Pod对象with-required-nodeaffinity的详细信息事件中已然出现成功调度至node01.ilinux.io节点的信息，具体如下所示：

```
Events:
  Type            Reason            Age           From           Message
  ----            -
Warning FailedScheduling 1m (x15 over 4m) default-scheduler 0/4 nodes
are available: 4 node(s) didn't match node selector.
Normal Scheduled 14s default-scheduler Successfully
assigned with-required-nodeaffinity to node01.ilinux.io
```

在定义节点亲和性时，requiredDuringSchedulingIgnoredDuringExecution字段的值是一个对象列表，用于定义节点硬亲和性，它可由一到多个nodeSelectorTerm定义的对象组成，彼此间为“逻辑或”的关系，进行匹配度检查时，在多个nodeSelectorTerm之间只要满足其中之一即可。nodeSelectorTerm用于定义节点选择器条目，其值为对象列表，它可由一个或多个matchExpressions对象定义的匹配规则组成，多个规则彼此之间为“逻辑与”的关系，这就意味着某节点的标签需要完全匹配同一个nodeSelectorTerm下所有的matchExpression对象定义的规则才算成功通过节点选择器条目的检查。而matchExmpressions又可由一到多个标签选择器组成，多个标签选择器彼此间为“逻辑与”的关系。

下面的资源配置清单示例（`required-nodeAffinity-pod2.yaml`）中定义了调度拥有两个标签选择器的节点挑选条目，两个标签选择器彼此之间为“逻辑与”的关系，因此，满足其条件的节点为`node01`和`node03`，如图12-5右侧的Pod对象的指向所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-required-nodeaffinity-2
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - {key: zone, operator: In, values: ["foo", "bar"]}
              - {key: ssd, operator: Exists, values: []}
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
```

构建标签选择器表达式中支持使用操作符有`In`、`NotIn`、`Exists`、`DoesNotExist`、`Lt`和`Gt`等，具体的用法请参考第4章中关于标签选择器的介绍。

另外，调度器在调度Pod资源时，节点亲和性（`MatchNodeSelector`）仅是其节点预选策略中遵循的预选机制之一，其他配置使用的预选策略依然正常参与节点预选过程。例如将上面资源配置清单示例中定义的Pod对象容器修改为如下内容并进行测试：

```
containers:
  - name: myapp
    image: ikubernetes/myapp:v1
    resources:
      requests:
        cpu: 6
        memory: 20Gi
```

在预选策略`PodFitsResources`根据节点资源可用性进行节点预选的过程中，它会获取给定节点的可分配资源量（资源问题减去已被运行于其上的各Pod对象的`requests`属性之和），去除那些无法容纳新Pod对象请求的资源量的节点。本书试验环境中使用的四个节点（其中一个为Master）配置相同，均为8核心CPU和16GB内存，它们都无法满足容器`myapp`的需求，因此调度失败，Pod资源会被置于`Pending`状态。下面

是将资源创建于集群中，而后通过其详细信息获取到的事件，它表明集群中仅有两个节点符合节点选择器，但4个节点都不具有充足的内存资源从而导致调度失败：

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Warning	FailedScheduling	4s (x2 over 4s)	default-scheduler	0/4 nodes are available: 2 node(s) didn't match node selector, 4Insufficient memory.

由上述操作过程可知，节点硬亲和性实现的功能与节点选择器（`nodeSelector`）相似，但亲和性支持使用匹配表达式来挑选节点，这一点提供了灵活且强大的选择机制，因此可被理解为新一代的节点选择器。

12.2.2 节点软亲和性

节点软亲和性为节点选择机制提供了一种柔性控制逻辑，被调度的Pod对象不再是“必须”而是“应该”放置于某些特定节点之上，当条件不满足时，它也能够接受被编排于其他不符合条件的节点之上。另外，它还为每种倾向性提供了weight属性以使用户定义其优先级，取值范围是1~100，数字越大优先级越高。下面一个Deployment资源配置清单示例（deploy-with-preferred-nodeAffinity.yaml）：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy-with-node-affinity
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 60
              preference:
                matchExpressions:
                  - {key: zone, operator: In, values: ["foo"]}
            - weight: 30
              preference:
                matchExpressions:
                  - {key: ssd, operator: Exists, values: []}
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
```

示例中，Pod资源模板定义了节点软亲和性以选择运行在拥有zone=foo和ssd标签（无论其值为何）的节点之上，其中zone=foo是更为重要的倾向性规则，它的权重为60，相比较来说，ssd标签就没有那么关键，它的权重为30。这么一来，如果集群中拥有足够多的节点，那么它将被此规则分为四类：同时满足拥有zone=foo和ssd标签、仅具有zone=foo标签、仅具有ssd标签，以及不具备此两个标签，如图12-6所示。

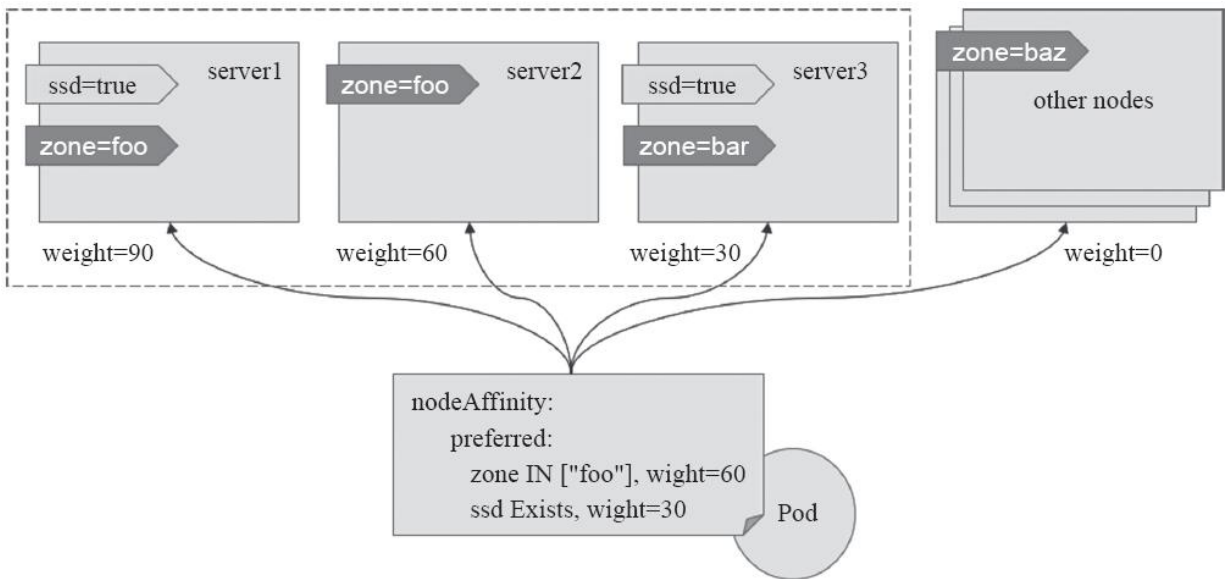


图12-6 节点软亲和性

以本书所用的测试环境为例，它共有三个节点（图12-6虚线内的节点），相对于myapp-deploy-with-node-affinity中定义节点亲和性规则来说，它们所拥有的倾向性权重分别如图12-6中标识的信息所示。在创建需要3个Pod对象的副本时，它们会不会被创建于同一节点node01之上？下面来验证其运行效果：

```
~]$ kubectl create -f deploy-with-preferred-nodeAffinity.yaml
deployment.apps "myapp-deploy-with-node-affinity" created
~]$ kubectl get pods -l app=myapp -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myapp-deploy-...-8qrv7	1/1	Running	0	12s	10.244.2.131	node02.ilinux.io
myapp-deploy-...-j5d7j	1/1	Running	0	12s	10.244.1.20	node01.ilinux.io
myapp-deploy-...-twdsr	1/1	Running	0	12s	10.244.3.149	node03.ilinux.io

结果显示，三个Pod对象被分散运行于集群中的三个节点之上，而非集中运行于node01节点。之所以如此，是因为使用了节点软亲和性的预选方式，所有节点均能够通过调度器上MatchNodeSelector预选策略的筛选，因此，可用节点取决于其他预选策略的筛选结果。在第二阶段的优选过程中，除了NodeAffinityPriority优选函数之外，还有其他几个优选函数参与优先级评估，尤其是SelectorSpreadPriority，它会将同一个ReplicaSet控制器管控的所有Pod对象分散到不同的节点上运行以抵御节点故障带来的风险。不过，这种节点亲和性的权重依然在发挥作用，如果把副本数量扩展至越过节点数很多（如15个），那么它

们将被调度器以接近节点亲和性权重比值（90： 60： 30）的方式分置于相关的节点之上，读者可自行验证其效果。

12.3 Pod资源亲和调度

出于高效通信的需求，偶尔需要把一些Pod对象组织在相近的位置（同一节点、机架、区域或地区等），如某业务的前端Pod和后端Pod等，此时可以将这些Pod对象间的关系称为亲和性。偶尔，出于安全或分布式等原因也有可能需要将一些Pod对象在其运行的位置上隔离开来，如在每个区域运行一个应用代理Pod对象等，此时可把这些Pod对象间的关系称为反亲和性（anti-affinity）。

当然，也可以通过节点亲和性来定义Pod对象间的亲和或反亲和特性，但用户必须为此明确指定Pod可运行的节点标签，显然这并非较优的选择。较理想的实现方式是，允许调度器把第一个Pod放置于任何位置，而后与其有亲和或反亲和关系的Pod据此动态完成位置编排，这就是Pod亲和性调度和反亲和性调度的功用。Pod的亲和性定义也存在“硬”（required）亲和性和“软”（preferred）亲和性的区别，它们表示的约束意义同节点亲和性相似。

Kubernetes调度器通过内建的MatchInterPodAffinity预选策略为这种调度方式完成节点预选，并基于InterPodAffinityPriority优选函数进行各节点的优选级评估。

12.3.1 位置拓扑

Pod亲和性调度需要各相关的Pod对象运行于“同一位置”，而反亲和性调度则要求它们不能运行于“同一位置”。何谓同一位置？事实上，它们取决于节点的位置拓扑，拓扑的方式不同，对于如图12-7中所示的Pod-A和Pod-B是否在同一位置的判定结果也可能有所不同。

如果以基于各节点的kubernetes.io/hostname标签作为评判标准，那么很显然，“同一位置”意味着同一个节点，不同节点即不同的位置，如图12-8所示。

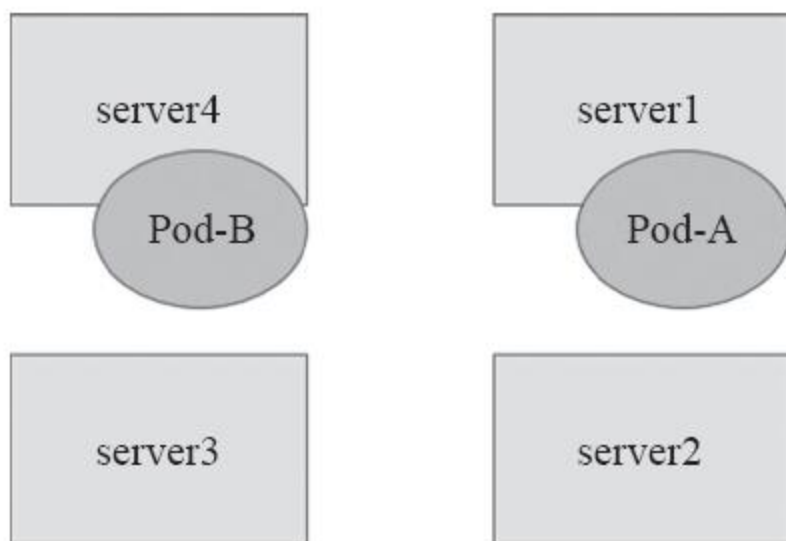


图12-7 Pod资源与位置拓扑

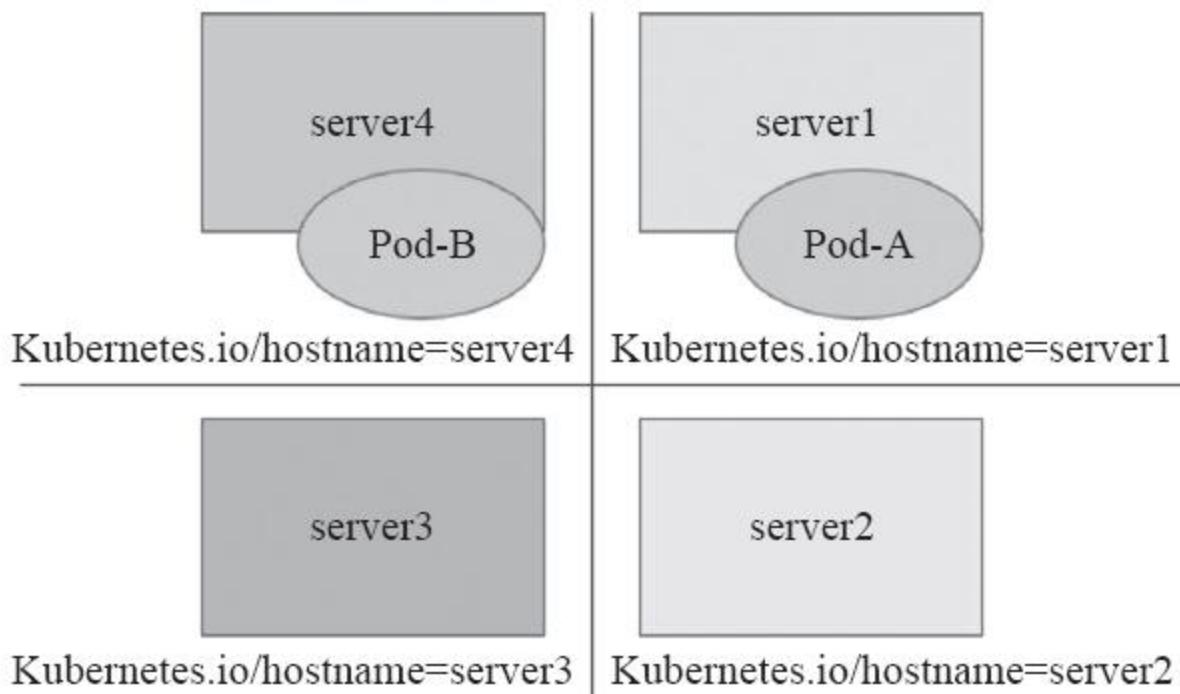


图12-8 基于节点的位置拓扑

而如果是基于如图12-9所划分的故障转移域来进行评判，那么server1和server4属于同一位置，而server2和server3属于另一个意义上的同一位置。

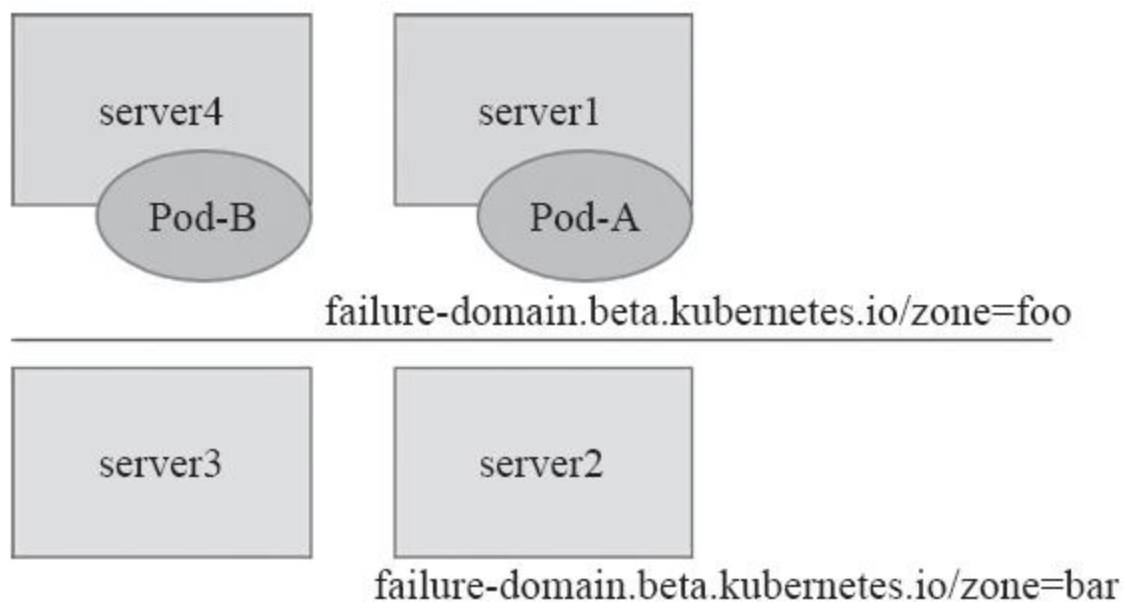


图12-9 基于故障转移域的位置拓扑

故此，在定义Pod对象的亲和性与反亲和性时，需要借助于标签选择器来选择被依赖的Pod对象，并根据选出的Pod对象所在节点的标签来判定“同一位置”的具体意义。

12.3.2 Pod硬亲和调度

Pod强约束的亲和性调度也使用 `requiredDuringSchedulingIgnoredDuringExecution` 属性进行定义。Pod亲和性用于描述一个Pod对象与具有某特征的现存Pod对象运行位置的依赖关系，因此，测试使用Pod亲和性约束，需要事先存在被依赖的Pod对象，它们具有特别的识别标签。下面创建一个有着标签“`app=tomcat`”的Deployment资源部署一个Pod对象：

```
~]$kubectl run tomcat-1 app=tomcat--image tomcat: alpine
```

下面的资源配置清单（`required-podAffinity-pod1.yaml`）中定义了一个Pod对象，它通过`labelSelector`定义的标签选择器挑选感兴趣的现存Pod对象，而后根据挑选出的Pod对象所在节点的标签“`kubernetes.io/hostname`”来判断同一位置的具体含义，并将当前Pod对象调度至这一位置的某节点之上：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity-1
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - {key: app, operator: In, values: ["tomcat"]}
          topologyKey: kubernetes.io/hostname
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
```

事实上，`kubernetes.io/hostname`标签是Kubernetes集群节点的内建标签，它的值为当前节点的节点主机名称标识，对于各个节点来说，各有不同。因此，新建的Pod对象将被部署至被依赖的Pod对象的同一节点之上，`requiredDuringSchedulingIgnoredDuringExecution`表示这种亲和性为强约束。

```
~]$ kubectl apply -f required-podAffinity-pod1.yaml
~]$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
tomcat-69f99cdf9d-6hvb5            1/1     Running   0           6m    10.244.3.152  node03.ilinux.io
with-pod-affinity-1                1/1     Running   0           4s    10.244.3.154  node03.ilinux.io
```

基于单一节点的Pod亲和性只在极个别的情况下才有可能用到，较为常用的通常是基于同一地区（**region**）、区域（**zone**）或机架（**rack**）的拓扑位置约束。例如部署应用程序服务（**myapp**）与数据库（**db**）服务相关的Pod时，**db** Pod可能会部署于如图12-10所示的foo或bar这两个区域中的某节点之上，依赖于数据服务的**myapp** Pod对象可部署于**db** Pod所在区域内的节点上。当然，如果**db** Pod在两个区域foo和bar中各有副本运行，那么**myapp** Pod将可以运行于这两个区域的任何节点之上。

例如，创建具有两个拥有标签为“**app=db**”的副本Pod作为被依赖的资源，它们可能运行于类似图12-5所示的三个节点中的任何一个或两个节点之上，本示例中它们凑巧运行于两个**zone**标签值不同的节点之上：

```
~]$ kubectl run db -l app=db --image=redis:alpine --replicas=2
~]$ kubectl get pods -l app=db -o wide -w
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
db-6b97b54df7-gbsl6               1/1     Running   .....           db-6b97b54df7-rhazp
db-6b97b54df7-rhazp               1/1     Running   .....           node02.ilinux.io
```

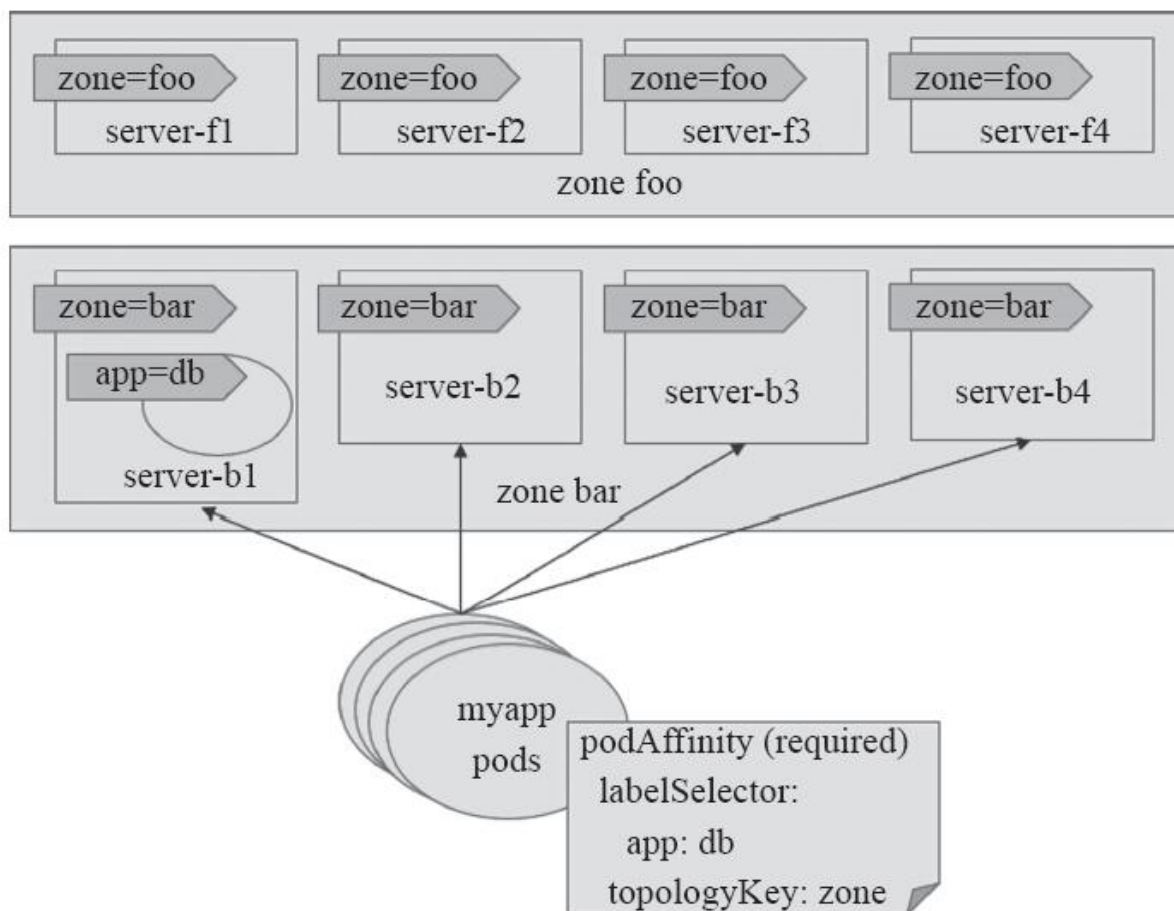


图12-10 Pod硬亲和性调度

于是，依赖于亲和于这两个Pod的其他Pod对象可运行于zone标签值为foo和bar的区域内的所有节点之上。下面的资源配置清单（`deploy-with-required-podAffinity.yaml`）中正是定义了这样一些Pod资源，它们由Deployment控制器所创建：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-with-pod-affinity
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
    spec:
```

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - {key: app, operator: In, values: ["db"]}
        topologyKey: zone
containers:
  - name: myapp
    image: ikubernetes/myapp:v1
```

在调度示例中的**Deployment**控制器创建的**Pod**资源时，调度器首先会基于标签选择器查询拥有标签“**app=db**”的所有**Pod**资源，接着获取到它们分别所属的节点的**zone**标签值，接下来再查询拥有匹配这些标签值的所有节点，从而完成节点预选。而后根据优选函数计算这些节点的优先级，从而挑选出运行新建**Pod**对象的节点。

需要注意的是，如果节点上的标签在运行时发生了更改，以致它不再满足**Pod**上的亲和性规则，但该**Pod**还将继续在该节点上运行，因此它仅会影响新建的**Pod**资源；另外，**labelSelector**属性仅匹配与被调度器的**Pod**在同一名称空间中的**Pod**资源，不过也可以通过为其添加**namespace**字段以指定其他名称空间。

12.3.3 Pod软亲和调度

类似于节点亲和性机制，Pod也支持使用 `preferredDuringSchedulingIgnoredDuringExecution` 属性定义柔性亲和机制，调度器会尽力确保满足亲和约束的调度逻辑，然而在约束条件不能得到满足时，它也允许将Pod对象调度至其他节点运行。下面是一个使用了Pod软亲和性调度机制的资源配置清单示例（`deploy-with-preferred-podAffinity.yaml`）：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-with-preferred-pod-affinity
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
    spec:
      affinity:
        podAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 80
              podAffinityTerm:
                labelSelector:
                  matchExpressions:
                    - {key: app, operator: In, values: ["cache"]}
                topologyKey: zone
            - weight: 20
              podAffinityTerm:
                labelSelector:
                  matchExpressions:
                    - {key: app, operator: In, values: ["db"]}
                topologyKey: zone
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
```

它定义了两组亲和性判定机制，一个是选择cache Pod所在节点的zone标签，并赋予了较高的权重80，另一个是选择db Pod所在节点的zone标签，它有着略低的权重20。于是，调度器会将目标节点分为四

类： cache Pod和db Pod同时所属的zone、cache Pod单独所属的zone、db Pod单独所属的zone， 以及其他所有的zone。

12.3.4 Pod反亲和调度

`podAffinity`用于定义Pod对象的亲和约束，对应地，将其替换为`podAntiAffinity`即可用于定义Pod对象的反亲和约束。不过，反亲和性调度一般用于分散同一类应用的Pod对象等，也包括将不同安全级别的Pod对象调度至不同的区域、机架或节点等。下面的资源配置清单（`deploy-with-required-podAntiAffinity.yaml`）中定义了由同一Deployment创建但彼此基于节点位置互斥的Pod对象：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-with-pod-anti-affinity
spec:
  replicas: 4
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - {key: app, operator: In, values: ["myapp"]}
              topologyKey: kubernetes.io/hostname
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
```

由于定义的强制性反亲和约束，因此，创建的4个Pod副本必须运行于不同的节点中。不过，本集群中一共只存在3个节点，因此，必然地会有一个Pod对象处于Pending状态，如下所示：

```
~]$ kubectl get pods -o wide -l app=myapp
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myapp-...-4gcvv	0/1	Pending	0	2s	<none>	<none>
myapp-...-788k6	1/1	Running	0	node01.ilinux.io	
myapp-...-cxqv5	1/1	Running	0	node03.ilinux.io	
myapp-...-mk28s	1/1	Running	0	node02.ilinux.io	

类似地，Pod反亲和性调度也支持使用柔性约束机制，在调度时，它将尽量满足不把位置相斥的Pod对象调度于同一位置，但是，当约束关系无法得到满足时，也可以违反约束而调度。读者可参考podAffinity的柔性约束示例将上面的Deployment资源myapp-with-pod-anti-affinity修改为柔性约束并进行调度测试。

12.4 污点和容忍度

污点（**taints**）是定义在节点之上的键值型属性数据，用于让节点拒绝将**Pod**调度运行于其上，除非该**Pod**对象具有接纳节点污点的容忍度。而容忍度（**tolerations**）是定义在**Pod**对象上的键值型属性数据，用于配置其可容忍的节点污点，而且调度器仅能将**Pod**对象调度至其能够容忍该节点污点的节点之上，如图12-11所示。

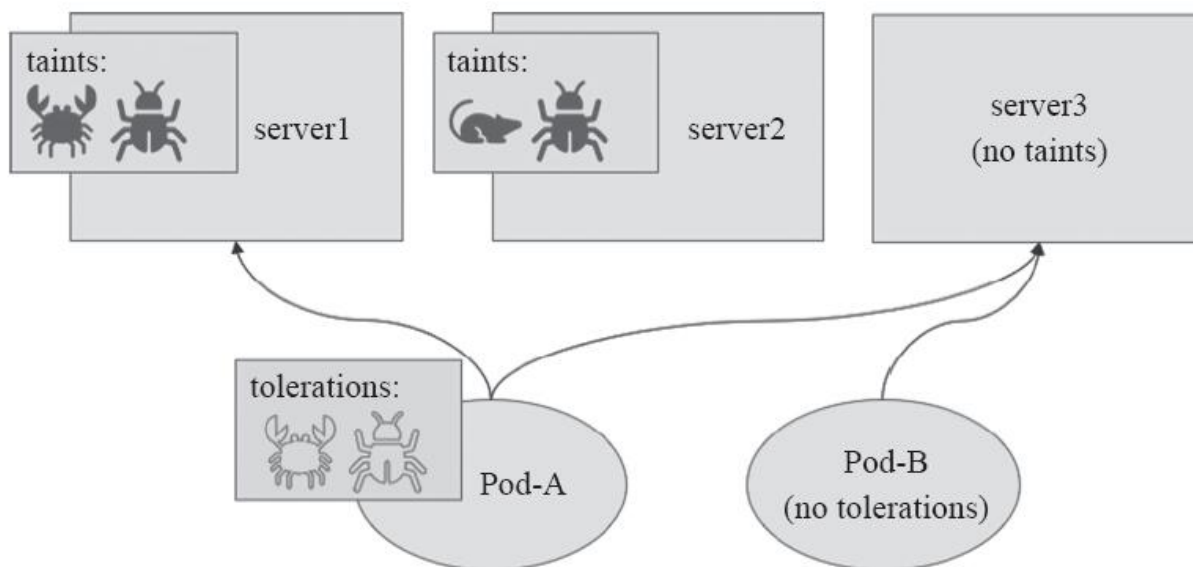


图12-11 污点与容忍之间的关系示意图

前文中，节点选择器（**nodeSelector**）和节点亲和性（**nodeAffinity**）两种调度方式都是通过在**Pod**对象上添加标签选择器来完成对特定类型节点标签的匹配，它们实现的是由**Pod**选择节点的机制。而污点和容忍度则是通过向节点添加污点信息来控制**Pod**对象的调度结果，从而赋予了节点控制何种**Pod**对象能够调度于其上的主控权。简单来说，节点亲和性使得**Pod**对象被吸引到一类特定的节点，而污点则相反，它提供了让节点排斥特定**Pod**对象的能力。

Kubernetes使用**PodToleratesNodeTaints**预选策略和**TaintTolerationPriority**优选函数来完成此种类型的高级调度机制。

12.4.1 定义污点和容忍度

污点定义在节点的`nodeSpec`中，而容忍度则定义在Pod的`podSpec`中，它们都是键值型数据，但又都额外支持一个效果（`effect`）标记，语法格式为“`key=value: effect`”，其中`key`和`value`的用法及格式与资源注解信息相似，而`effect`则用于定义对Pod对象的排斥等级，它主要包含以下三种类型。

·**NoSchedule**: 不能容忍此污点的新Pod对象不可调度至当前节点，属于强制型约束关系，节点上现存的Pod对象不受影响。

·**PreferNoSchedule**: **NoSchedule**的柔性约束版本，即不能容忍此污点的新Pod对象尽量不要调度至当前节点，不过无其他节点可供调度时也允许接受相应的Pod对象。节点上现存的Pod对象不受影响。

·**NoExecute**: 不能容忍此污点的新Pod对象不可调度至当前节点，属于强制型约束关系，而且节点上现存的Pod对象因节点污点变动或Pod容忍度变动而不再满足匹配规则时，Pod对象将被驱逐。

此外，在Pod对象上定义容忍度时，它支持两种操作符：一种是等值比较（**Equal**），表示容忍度与污点必须在`key`、`value`和`effect`三者之上完全匹配；另一种是存在性判断（**Exists**），表示二者的`key`和`effect`必须完全匹配，而容忍度中的`value`字段要使用空值。

另外，一个节点可以配置使用多个污点，一个Pod对象也可以有多个容忍度，不过二者在进行匹配检查时应遵循如下逻辑。

- 1) 首先处理每个有着与之匹配的容忍度的污点。
- 2) 不能匹配到的污点上，如果存在一个污点使用了**NoSchedule**效用标识，则拒绝调度Pod对象至此节点。
- 3) 不能匹配到的污点上，若没有任何一个使用了**NoSchedule**效用标识，但至少有一个使用了**PreferNoScheduler**，则应尽量避免将Pod对象调度至此节点。

4) 如果至少有一个不匹配的污点使用了**NoExecute**效用标识，则节点将立即驱逐**Pod**对象，或者不予调度至给定节点；另外，即便容忍度可以匹配到使用了**NoExecute**效用标识的污点，若在定义容忍度时还同时使用**tolerationSeconds**属性定义了容忍时限，则超出时限后其也将被节点驱逐。

使用**kubeadm**部署的**Kubernetes**集群，其**Master**节点将自动添加污点信息以阻止不能容忍此污点的**Pod**对象调度至此节点，因此，用户手动创建的未特意添加容忍此污点容忍度的**Pod**对象将不会被调度至此节点：

```
~]$ kubectl describe node master.ilinux.io
Name:                master.ilinux.io
Roles:               master
Labels:              beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=master.ilinux.io
                    node-role.kubernetes.io/master=
.....
Taints:              node-role.kubernetes.io/master:NoSchedule
Unschedulable:      false
```

不过，有些系统级应用，如**kube-proxy**或者**kube-flannel**等，都在资源创建时就添加上了相应的容忍度以确保它们被**DaemonSet**控制器创建时能够调度至**Master**节点运行一个实例：

```
~]$ kubectl describe pods kube-flannel-ds-lsgqr -n kube-system
.....
Node-Selectors:      beta.kubernetes.io/arch=amd64
Tolerations:         node-role.kubernetes.io/master:NoSchedule
                    node.kubernetes.io/disk-pressure:NoSchedule
                    node.kubernetes.io/memory-pressure:NoSchedule
                    node.kubernetes.io/not-ready:NoExecute
                    node.kubernetes.io/unreachable:NoExecute
```

另外，这类**Pod**是构成**Kubernetes**系统的基础且关键性的组件，它们甚至还定义了更大的容忍度。从上面某**kube-flannel**实例的容忍度定义来看，它还能容忍那些报告了磁盘压力或内存压力的节点，以及未就绪的节点和不可达的节点，以确保它们能在任何状态下正常调度至集群节点上运行。

12.4.2 管理节点的污点

任何符合其键值规范要求的字符串均可用于定义污点信息：仅可使用字母、数字、连接符、点号和下划线，且仅能以字母或数字开头，其中键名的长度上限为253个字符，值最长为63个字符。实践中，污点通常用于描述具体的部署规划，它们的键名形如node-type、node-role、node-project或node-geo等，因此还可在必要时带上域名以描述其额外的信息，如node-type.ilinux.io等。使用“`kubectl taint`”命令即可向节点添加污点，命令的语法格式如下：

```
kubectl taint nodes <node-name> <key>=<value>:<effect> ...
```

例如，使用“node-type=production: NoSchedule”定义节点node01.ilinux.io：

```
~]$ kubectl taint nodes node01.ilinux.io node-type=production:NoSchedule
node "node01.ilinux.io" tainted
```

此时，node01上已有的Pod对象不受影响，但新建的Pod若不能容忍此污点将不能再被调度至此节点。类似下面的命令可以查看节点上的污点信息：

```
~]$ kubectl get nodes node01.ilinux.io -o go-template={{.spec.taints}}
[map[value:production effect:NoSchedule key:node-type]]
```

需要注意的是，即便是同一个键值数据，若其效用标识不同，则其也分属于不同的污点信息，例如，将上面命令中的效用标识定义为PreferNoSchedule再添加一次：

```
~]$ kubectl taint nodes node01.ilinux.io node-type=production:PreferNoSchedule
node "node01.ilinux.io" tainted
```

删除某污点，仍然通过**kubectl taint**命令进行，但要使用如下的命令格式，省略效用标识则表示删除使用指定键名的所有污点，否则就只删除指定键名上对应效用标识的污点：

```
kubectl taint nodes <node-name> <key>[:<effect>]-
```

例如，删除node01上node-type键的效用标识为“NoSchedule”的污点信息：

```
~]$ kubectl taint nodes node01.ilinux.io node-type:NoSchedule-  
node "node01.ilinux.io" untainted
```

若要删除使用指定键名的所有污点，则在删除命令中省略效用标识即能实现，例如：

```
~]$ kubectl taint nodes node01.ilinux.io node-type-  
node "node01.ilinux.io" untainted
```

删除节点上的全部污点信息，通过**kubectl patch**命令将节点属性spec.taints的值直接置空即可，例如：

```
~]$ kubectl patch nodes node01.ilinux.io -p '{"spec":{"taints":[]}}'  
node "node01.ilinux.io" patched
```

节点污点的变动会影响到新建Pod对象的调度结果，而且使用NoExecute进行标识时，还会影响到节点上现有的Pod对象。

12.4.3 Pod对象的容忍度

Pod对象的容忍度可通过其spec.tolerations字段进行添加，根据使用的操作符不同，主要有两种可用的形式：一种是与污点信息完全匹配的等值关系；另一种是判断污点信息存在性的匹配方式。使用Equal操作符的示例如下所示，其中tolerationSeconds用于定义延迟驱逐当前Pod对象的时长：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

使用存在性判断机制的容忍度示例如下所示：

```
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

实践中，若集群中的一组机器专用于为运行非生产型的容器应用而备置，而且它们可能随时按需上下线，那么就应该为其添加污点信息，以确保仅那些能容忍此污点的非生产型Pod对象可以调度其上。另外，某些有着特殊硬件的节点需要专用于运行一类有着此类硬件资源需求的Pod对象时，例如，那些有着SSD或GPU的设备，也应该为其添加污点信息以排除其他的Pod对象。

12.4.4 问题节点标识

Kubernetes自1.6版本起支持使用污点自动标识问题节点，它通过节点控制器在特定条件下自动为节点添加污点信息实现。它们都使用NoExecute效用标识，因此不能容忍此类污点的现有Pod对象也会遭到驱逐。目前，内建使用的此类污点包含如下几个。

- node.kubernetes.io/not-ready: 节点进入“NotReady”状态时被自动添加的污点。

- node.alpha.kubernetes.io/unreachable: 节点进入“NotReachable”状态时被自动添加的污点。

- node.kubernetes.io/out-of-disk: 节点进入“OutOfDisk”状态时被自动添加的污点。

- node.kubernetes.io/memory-pressure: 节点内存资源面临压力。

- node.kubernetes.io/disk-pressure: 节点磁盘资源面临压力。

- node.kubernetes.io/network-unavailable: 节点网络不可用。

- node.cloudprovider.kubernetes.io/uninitialized: kubelet由外部的云环境程序启动时，它将自动为节点添加此污点，待到云控制器管理器中的控制器初始化此节点时再将其删除。

不过，Kubernetes的核心组件通常都要容忍此类的污点，以确保其相应的DaemonSet控制器能够无视此类污点，于节点上部署相应的关键性Pod对象，例如kube-proxy或kube-flannel等。

12.5 Pod优先级和抢占式调度

Kubernetes自1.8版本起开始支持Pod资源的优先级机制，它用于表现一个Pod对象相对于其他Pod对象的重要程度。一个Pod对象无法被调度时，调度器会尝试抢占（驱逐）较低优先级的Pod对象，以便可以调度当前Pod。另外，在Kubernetes 1.9和更高版本中，优先级还会影响节点上Pod的调度顺序和驱逐次序。

不过，Pod优先级和抢占机制默认处于禁用状态，如需启用，则需要同时为kube-apiserver、kube-scheduler和kubelet程序的“**--feature-gates**”选项添加“**PodPriority=true**”条目。添加完成后，事先创建好优先级类别，并在创建Pod资源时通过**priorityClassName**属性指定其所属的优先级类别即可。

此种特性目前仍处于Alpha阶段，因此，本章不再对其进行过多的描述，感兴趣的读者可参考文档进行测试。

12.6 本章小结

本章讲解了Kubernetes默认调度器的分步调度机制，描述了各预选策略和优选函数的功能，并对节点亲和性调度、Pod亲和性调度以及基于污点和容忍的高级调度方式进行了重点说明。

第13章 Kubernetes系统扩展

Kubernetes系统的扩展和增强既包括扩展API Server所支持的资源类型及相关声明式功能的实现，以及消除集群的单点以实现集群的高可用等，也包括如何将系统增强为一个完整意义上的PaaS平台，并以DevOps文化为驱动改善工作流程等。

13.1 自定义资源类型（CRD）

Kubernetes API默认提供的众多的功能性资源类型可用于容器编排以解决多数场景中的编排需求。然而，有些场景也许要借助于额外的或更高级别的编排抽象，例如引入集群外部的一些服务并以资源对象的形式进行管理，又或者把Kubernetes的多个标准资源对象合并为一个单一的更高级别的资源抽象，等等。而这类API扩展抽象通常也应该兼容Kubernetes系统的基本特性，如支持kubectl管理工具、CRUD及watch机制、标签、etcd存储、认证、授权、RBAC及审计，等等，从而使得用户可将精力集中于构建业务逻辑本身。

目前，扩展Kubernetes API的常用方式有三种：使用CRD（CustomResourceDefinitions）自定义资源类型、开发自定义的API Server并聚合至主API Server，以及定制扩展Kubernetes源码。其中，CRD最为易用但限制颇多，自定义API Server更富于弹性但代码工作量偏大，而仅在必须添加新的核心类型才能确保专用的Kubernetes集群功能正常时才应该定制系统源码。

在某种程度上，Kubernetes API Server可以看作一个JSON方案的数据库存储系统，它内建了众多数据模式（资源类型），以etcd为存储后端，支持存储和检索结合内建模式进行实例化的数据项（对象）。作为客户端，控制器会收到有关这些资源对象变动的通知，并在响应过程中操纵这些对象及相关的其他资源，或者将这些更改反映到外部系统（如云端的软件负载平衡器）。从这个角度进行类比，CRD就像是由用户为Kubernetes存储系统提供的自定义数据模式，基于这些模式进行实例化的数据项一样可以存入系统中。

CRD并非设计用来取代Kubernetes的原生资源类型，而是用于补充一种简单易用的更为灵活和更高级别的自定义API资源的方式。虽然目前在功能上仍存在不少的局限，但对于大多数的需求场景来说，CRD的表现已经足够好，因此在满足需求的前提下是首选的API资源类型扩展方案。

13.1.1 创建CRD对象

CRD自Kubernetes 1.7版开始引入，并自1.8版起完全取代其前身TPR（ThirdParty-Resources），其设计目标是无须修改Kubernetes源代码就能扩展它支持使用API资源类型。CRD本身也是一种资源类型，隶属于集群级别，实例化出特定的对象之后，它会在API上注册生成GVR类型URL端点，并能够作为一种资源类型被使用并实例化相应的对象。自定义资源类型之前，选定其使用的API群组名称、版本及新建的资源类型名称，根据这些信息即可创建自定义资源类型，并创建自定义类型的资源对象，其流程如图13-1所示。

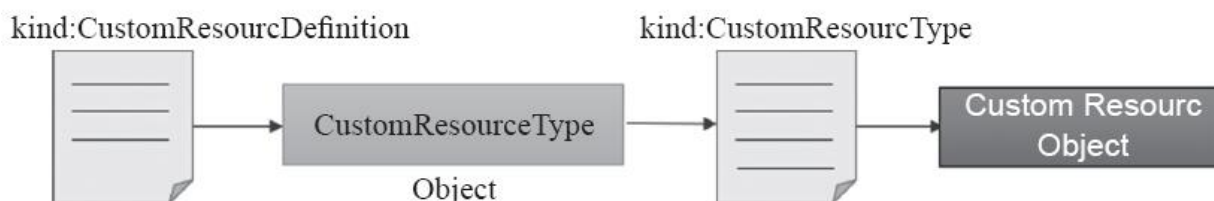


图13-1 创建自定义资源类型及自定义类型的资源对象

下面的配置清单中定义了一个名为users.auth.ilinux.io的CRD资源对象，它的群组名称为auth.ilinux.io，仅支持一个版本级别v1beta1，复数形式为users，隶属于名称空间级别，因此，它的对象在API上的URL路径前缀为/apis/auth.ilinux.io/v1beta1/namespace/NS_NAME/users/：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: users.auth.ilinux.io
spec:
  group: auth.ilinux.io
  version: v1beta1
  names:
    kind: User
    plural: users
    singular: user
    shortNames:
      - u
  scope: Namespaced
```

配置清单中，spec嵌套使用的字段都能够见名知义，这里需要特别说明的是，metadata.name字段的值必须等同于spec字段中的“<names.plural>.<group>”合并起来的形式。将清单中的CRD资源创建于Kubernetes集群中之后，继而创建一个新的CustomResourceDefinition类型的对象，例如，使用下面的命令列出集群上的CRD对象时，命令结果将显示出如下资源名称及创建时间状态信息：

```
~]$ kubectl get crd
NAME                               CREATED AT
users.auth.ilinux.io             2018-08-24T12:51:54Z
```

现在，在API群组auth.ilinux.io/v1beta1中，users已经是一个名称空间级别的可用资源类型，用户可按需创建出任意数量的users类型的对象，下面就是一个资源清单示例，它定义了一个名为admin的users对象：

```
apiVersion: auth.ilinux.io/v1beta1
kind: User
metadata:
  name: admin
  namespace: default
spec:
  userID: 1
  email: k8s@ilinux.io
  groups:
  - superusers
  - administrators
  password: ikubernetes
```

自定义资源类型users并未限制其spec字段中可嵌套使用的字段及其数据类型，于是用户可按需随意使用任何字段名及字段值，例如上面的资源清单中使用的userID、email和groups等字段。资源创建完成后即可使用users作为类型标识使用kubectl命令完成资源对象的管理，包括查看、删除、修改等操作。例如，将清单中的自定义资源创建于名称空间中，而后使用类似如下的命令获取相关的状态信息：

```
~]$ kubectl get users -n default
NAME    AGE
admin   10s
```

根据API对象GVR格式的URL规范，users资源的对象admin的引用路径为/apis/auth.ilinux.io/v1beta1/namespaces/default/users/admin，这一点可以通过describe命令予以证实。而要删除自定义的users对象admin，只要使用通用格式的kubectl命令即可，如“kubectl delete users admin”，或者使用陈述式对象配置命令“kubectl delete-f<filename>”。

13.1.2 自定义资源格式验证

除了对操作请求进行身份认证和授权检查之外，对象配置的变动在存入etcd之前还需要经由准入控制器的核验，尤其是验证型

（**validation**）控制器会检查传入的对象格式是否符合有效格式，包括是否设定了不符合定义的数据类型值，以及是否违反了字段的限制规则等。

Kubernetes自1.9版本起支持为CRD定义验证（**validation**字段）机制用以定义CRD可用的有效字段等，这在将设计的自定义资源公开应用时非常重要。自定义资源可使用OpenAPI模式声明验证规则，该模式是JSON模式的子集。需要注意的是，OpenAPI架构并不能支持JSON架构的所有功能，而CRD验证也不能支持OpenAPI架构的所有功能，不过，对于大多数情况来说，它足够用了。



提示 CRD的**validation**中支持使用的限制机制、数据类型及数据格式等请参考其API手册。

下面是一个配置清单片段，将其合并添加到13.1.1节中CRD对象users.auth.ilinux.io配置清单的spec内即能生效。它分别定义了**userID**、**groups**、**email**和**password**字段的数据类型，并指定了**userID**字段的取值范围，以及**password**字段的数据格式，而且还通过**required**指定**userID**和**groups**是必选字段：

```
spec:
  validation:
    openAPIV3Schema:
      properties:
        spec:
          properties:
            userID:
              type: integer
              minimum: 1
              maximum: 65535
            groups:
              type: array
            email:
              type: string
            password:
              type: string
```

```
format: password
required: ["userID", "groups"]
```

限制了每个字段的数据类型之后，验证机制会检查创建或修改操作中提供的每个字段值等是否违反了格式要求，任何的核验失败都会导致写入操作被拒绝。下面的内容是某次创建操作返回的错误提示，它显示命令提供的对象配置信息违反了**userID**字段的取值范围，并且缺少必选字段：**spec.groups**。

```
The User "tony" is invalid: []: ...
...: validation failure list:
spec.userID in body should be less than or equal to 65535
spec.groups in body is required
```

自Kubernetes 1.11版本开始，**kubectl**即使用服务器侧对象信息打印机制，这意味着将由**API Server**决定**kubectl get**命令结果会显示哪些字段。在**CRD**资源中，可在**spec.additionalPrinterColumns**中嵌套定义在对象的详细信息中要打印的字段列表。下面的配置清单片段定义了要为**users**类型的对象显示相关的四个字段，将其合并至前面定义的**CRD**对象**users**配置清单的**spec**字段中，重新应用到集群中并确保其正常生效：

```
spec:
  additionalPrinterColumns:
    - name: userID
      type: integer
      description: The user ID.
      JSONPath: .spec.userID
    - name: groups
      type: string
      description: The groups of the user.
      JSONPath: .spec.groups
    - name: email
      type: string
      description: The email address of the user.
      JSONPath: .spec.email
    - name: password
      type: string
      description: The password of the user account.
      JSONPath: .spec.password
```

以上四个字段会显示在**get**、**describe**等命令查看**users**类型的对象状态信息的输出结果中：

```
~]$ kubectl get users admin
```

NAME	USERID	GROUPS	EMAIL	PASSWORD
admin	1	[superusers administrators]	k8s@ilinux.io	ikubernetes

当然，在对象中以明文字符串保存密码并不是一个好的选择，真正用到时，应该使用**Secret**对象保存密钥信息，并在**users**对象中进行引用。

13.1.3 子资源

可能有读者已经注意到，前面自定义资源users的对象admin在其详细状态信息输出中没有类似核心资源的status字段，该字段是一种用于保存对象当前状态的子资源。在Kubernetes系统的声明式API中，status字段至关重要，它由Kubernetes系统自行维护，相关的控制器在和解循环中持续与API Server进行通信，并负责确保status字段中的状态匹配spec字段中定义的期望状态。

在Kubernetes 1.10版本之前，自定义资源的API端点不区分spec和status字段，而自1.10版本起，自定义资源开始支持通过/status子资源的方式提供对象的当前状态，虽然它仍然不会显示于获取状态信息的命令结果输出中，但客户端可通过对象的子URL路径来获取状态信息。此特性在1.11版本中已经升级至beta级别。

在CRD中为自定义资源启用status字段的方式非常简单，只需要为其定义spec.subresources.status字段即可，其内嵌的字段等由系统自行维护，用户无须提供任何额外的配置。它的使用格式如下：

```
spec:
  subresources:
    status: {}
```

将上面配置清单中的配置片段合并至前面创建的CRD对象users的配置中，并完成活动对象的修改即可在其实例化出的对象上通过/status获取状态信息，如对象admin的状态引用路径为/apis/auth.ilinux.io/v1beta1/namespaces/default/users/admin/status。不过，在没有相应资源控制器的情形下，活动对象状态的非计划内变动既不会实时反映到status中，也不会向spec定义的期望状态转换。

事实上，如果有相应的资源控制器维护自定义的资源类型时，还可以在配置自定义资源对象时使用scale子资源和status子资源协同实现类似Deployment或StatefulSet等控制器一样对象规模的伸缩功能。它的定义格式如下：

```
spec:
  status: {}
  scale:
    specReplicasPath:
    statusReplicasPath:
    labelSelectorPath:
```

需要注意的是，**scale**字段必须与**status**字段一起使用，由控制器通过**status**获取对象当前的副本数量，并与**spec**字段内嵌套的用于指定副本数量的字段（例如，常用的**replicas**）进行比较来确定其所需要执行的伸缩操作，而后再将伸缩操作的结果更新至**status**字段中。上面的配置格式中，**scale**的各内嵌字段功用说明具体如下。

·**specReplicasPath<string>**：引用定义在**spec**中用于表示期望的副本数量的字段，JSONPath格式，如**.spec.replicas**。

·**statusReplicasPath<string>**：引用保存在**status**中用于表示对象的当前副本数量的字段，JSONPath格式，如**.status.replicas**。

·**labelSelectorPath<string>**：可选字段，但若要与**HPA**结合使用则是必选字段，用于引用**status**中用于表示使用的标签选择器字段，JSONPath格式，如**.status.labelSelector**。

为某CRD资源定义**scale**子资源之后，即可实例化出支持规模伸缩的自定义资源对象。但必须存在一个相应的资源控制器来持续维护相应的自定义资源对象，以确保其当前状态匹配期望的状态。满足条件后，类似于**Deployment**资源对象等，使用“**kubectl scale**”命令就能完成对自定义资源对象规模的手动伸缩。

13.1.4 使用资源类别

类别（categories）是Kubernetes 1.10版本引入的一种分组组织自定义资源的方法，定义CRD对象时为其指定一个或多个类别，可以通过`kubectl get<category-name>`命令列出该类别中的所有自定义资源对象，`all`就是一个常用的内建资源类别。例如，为前面定义的CRD对象`users`的`spec.names`字段中额外内嵌如下配置，便能使得`users`资源类型下的所有对象都隶属于`all`类别：

```
spec:
  names:
    categories:
      - all
```

`categories`字段的值是自定义资源所属的分组资源列表。而后，创建的所有自定义类型`users`的对象都能够通过“`kubectl get all`”命令进行获取：

```
~]$ kubectl get all
.....
```

NAME	USERID	GROUPS	EMAIL
user.auth.ilinux.io/admin	1	[superusers administrators]	k8s@ilinux.io

13.1.5 多版本支持

Kubernetes原生资源的一个引人注目的特性是它们能够在API版本之间自动和透明地迁移。资源的使用者可以使用混合API版本，并且都能获得他们所期望的资源版本。而自Kubernetes 1.11版本开始，CRD支持多个版本，但它们之间的转换必须手动完成。

在CRD上使用多版本机制时，将spec.version字段替换为spec.versions字段，并将支持的各版本以对象列表的形式给出定义即可。不过，多版本并存时，仅其中一个版本且必须有一个版本应标记为存储（spec.versions[].storage字段）版本。下面的配置清单片段中定义了两个API版本，其中仅v1beta1标记为了storage：

```
spec:
  versions:
  - name: v1beta1
    served: true
    storage: true
  - name: v1beta2
    served: true
    storage: false
```

将上述配置清单版本并入此前定义的users资源配置清单中并应用于活动对象后再创建相应类型的自定义对象时就可以通过两个不同的API版本之一来完成。

13.1.6 自定义控制器基础

仅借助于CRD完成资源自定义本身并不能为用户带来太多的价值，它只是资源类型的定义，只是提供了JSON格式的数据范式及存取相关数据的能力，至于如何执行数据相关的业务逻辑，时刻确保将status中的状态移向spec中的状态则是由封装于控制器中的代码来负责实现的。相应地，为自定义资源类型提供业务逻辑代码的控制器需要由用户自行开发，并运行于API Server的客户端程序（通常是托管运行于Kubernetes系统之上，类似于Ingress控制器），这就是所谓的自定义控制器（Custom Controller）。换句话讲，某特定的CRD资源的相关对象发生变动时，如何确保它的当前状态不断地接近期望的状态并非API Server的功能，而是相关的专用控制器组件。

事实上，自定义控制器不仅仅能够用于管理CRD资源，用户也完全可以仅针对系统内建的核心类型对象开发更高级别的控制器，这些对象类型包括Service、Deployment、ConfigMap等。不过，对于每个CRD的自定义类型来说，至少应该存在一个相关的自定义控制器，如图13-2所示。

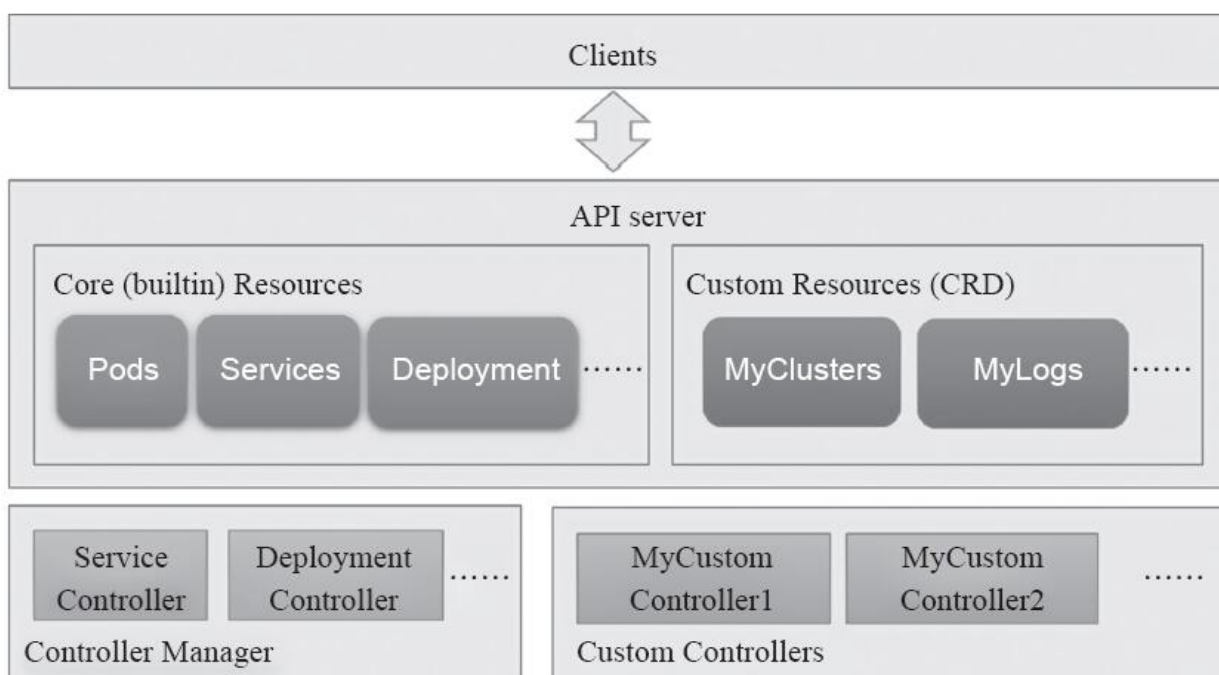


图13-2 Kubernetes核心资源与CRD

简单来说，控制器负责持续监视资源变动，根据资源的spec及status中的信息执行某些操作逻辑，并将执行结果更新至资源的status中。自定义控制器同样担负着类似的职责，只不过一个特定的控制器通常仅负责管理一部分特定的资源类型，并执行专有管理逻辑。

Kubernetes系统内建了许多控制器，如NodeController、ServiceController等，它们打包在一起并作为一个统一守护进程kube-controller-manager。这些控制器基本上都遵循同一种模式以完成资源管理操作，该模式通常可描述为三种特性：声明式API、异步和水平式处理。

- 声明式API：用户定义期望的状态而非要执行的特定操作，如使用kubectl apply命令将请求发送至API Server存储于etcd中，并由API Server做出处理响应。

- 异步：客户端的请求于API Server存储完成之后即返回成功信息，而无须等待控制器执行的和解循环结束。

- 水平式处理（level-based）：同一对象有多次变动事件待处理时，控制器仅需要处理其最新一次的变动。这种根据最新观察结果而非历史变化来和解status和spec的机制即为水平式处理机制。

自定义控制器实现自定义资源管理行为的最佳方法同样是遵循此类控制器模式进行程序开发，目前，大部分开发接口都是基于Golang语言来实现的，相关代码位于客户端库的client-go/go/tools/cache和client-go/util/workqueue目录中。

简单来说，控制器包含两个重要组件：Informer/SharedInformer和Workqueue，前者负责监视资源对象当前状态的更改，并将事件发送至后者，而后由处理函数进行处理，如图13-3所示。

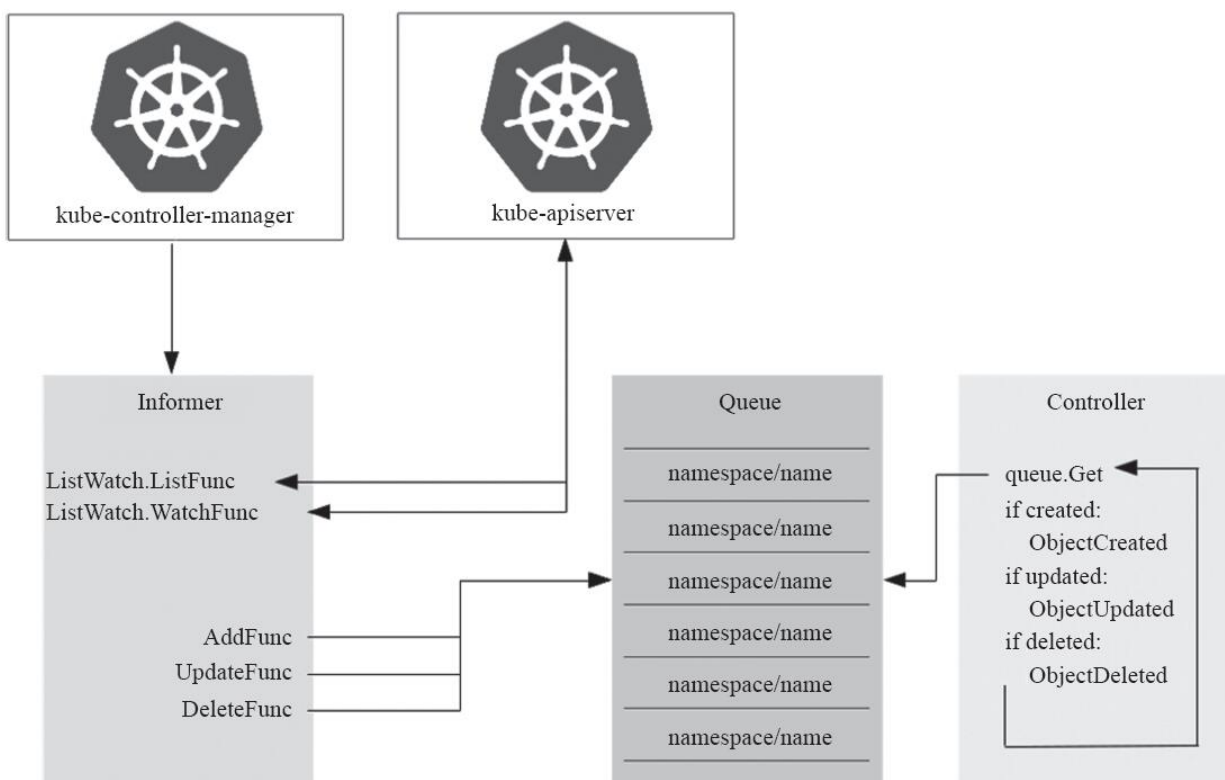


图13-3 处理器事件流动示意图（图片来源：<https://medium.com/@trstringer>）

Informer主要由Listwatcher、ResourceEventHandler和ResyncPeriod三类函数组件进行构造。

- Listwatcher是应用于特定名称空间中特定资源对象的列表函数（listFunc）和监视函数（watchFunc），并结合字段选择器为控制器精确限定关注的资源对象。

- ResourceEventHandler负责处理资源对象状态改变产生的相关通知，其分别使用AddFunc、UpdateFunc和DeleteFunc三个函数完成对象的创建、更新和删除操作。

- ResyncPeriod定义控制器遍历缓存中的所有项目并触发运行UpdateFunc的频率，即和解循环的执行频度。较高的频度对于控制器错过某次更新或此前的更新操作执行失败时非常有用，但会对系统资源带来较高的压力，因此具体的时长需要全面、系统地进行权衡。

Informer是控制器的私有组件，它为相关资源对象创建的缓存信息仅可供当前控制器使用。而在Kubernetes系统上，同一资源对象支持多

个控制器共同处理，而且多个控制器监视同一资源对象也是较为常见的情形，于是，一个更高效的方式是使用替代解决方案SharedInformer和Workqueue。SharedInformer支持在监控同类资源对象的控制器之间创建共享缓存，这有效降低了内存资源开销，而且它也仅需要在上游的API Server中注册创建一个监视器，即能显著减轻上游服务器的访问压力。因此，较之Informer，SharedInformer才是更为常用的解决方案。

不过，SharedInformer无法跟踪每个控制器的位置（因为它是共享的），于是控制器必须负责提供自用的工作队列及重试机制，这也意味着它的ResourceEventHandler程序只是将事件放在每个消费者的Workqueue中。

- 每当资源发生变化时，ResourceEventHandler程序都会将一个键放入工作队列中，对于名称空间级别资源对象的相关事件，其键名格式为<resource_namespace>/<resource_name>，集群级别资源对象的键名则只包含<resource_name>。

- 目前，工作队列存在延迟队列、定时队列和速率限制队列等几种形式。

因此，自定义控制器构建起来存在一定的复杂度。实践中，为了便于用户使用client-go创建控制器，Kubernetes社区发布了模板类的项目workqueue example和sample-controller，它们提供了一个自定义控制器项目应有的基础结构。通常的做法是复制相应的代码并按需修改相应的部分，例如，将syncHandler修改为自定义资源类型业务处理逻辑等，而后借助于code-generator项目用脚本完成相应的组件，如typed clients、informers等。虽然好过从零构建自定义控制器的代码，但这类方式总觉得有些不趁手和美中不足。

好在，现在已经有了几类更加成熟、更易上手的工具可用，它们甚至已经可以被视作开发CRD和控制器的SDK或框架，其中，主流的项目主要有Kubebuilder、Operator SDK和Metacontroller三个。

Kubebuilder主要由Google的工程师Phillip Wittrock创立，但目前归属于SIG API Machinery，有较完善的在线文档。Operator SDK是CoreOS发布的开源项目，是Operator Framework的一个子集，其出现时间略早，社区接受度较高，以致很多人干脆就把自定义控制器与Operator当作同一事物不加区分地使用，目前已经有etcd、Prometheus、Rook和Vault几个成熟的Operator可用。Metacontroller由GCP发布，与前两者的区别较

大，它将控制器模式直接委托给**Metacontroller**框架，并调用用户提供的**WebHook**实现模式的处理功能，支持任何编程语言开发，接收并返回**JSON**格式的序列化数据。

限于篇幅，这里就不再介绍它们各自的具体用法了，对自定义控制器有兴趣的读者可参考相关项目的文档进行学习。另外，有兴趣测试**CRD**及自定义控制器的读者也可以参考**GitHub**上的项目 **nikhita/custom-database-controller**，项目地址为 <https://github.com/nikhita/custom-database-controller> 。

13.2 自定义API Server

扩展Kubernetes系统API接口的另一种常用办法是使用自定义的API Server。相较于CRD来说，使用自定义API Server更加灵活，例如可以自定义资源类型和子资源、自定义验证及其他逻辑，甚至于在Kubernetes API Server中实现的任何功能也都能在自定义API Server中实现。

13.2.1 自定义API Server概述

自定义API Server完全可以独立运行并能够被客户端直接访问，但最方便的方式是将它们与主API Server（kube-apiserver）聚合在一起使用，这样不仅可以使得集群中的多个API Server看起来好像是由单个服务器提供服务，集群组件和kubectl等客户端可不经修改地继续正常通信，而且也无须特殊逻辑来发现不同的API Server等操作。在kube-apiserver中，用于聚合自定义API Server的组件是kube-aggregator，它自Kubernetes 1.7版本引入，并内建于主API Server之中作为其进程的一部分来运行，如图13-4所示。

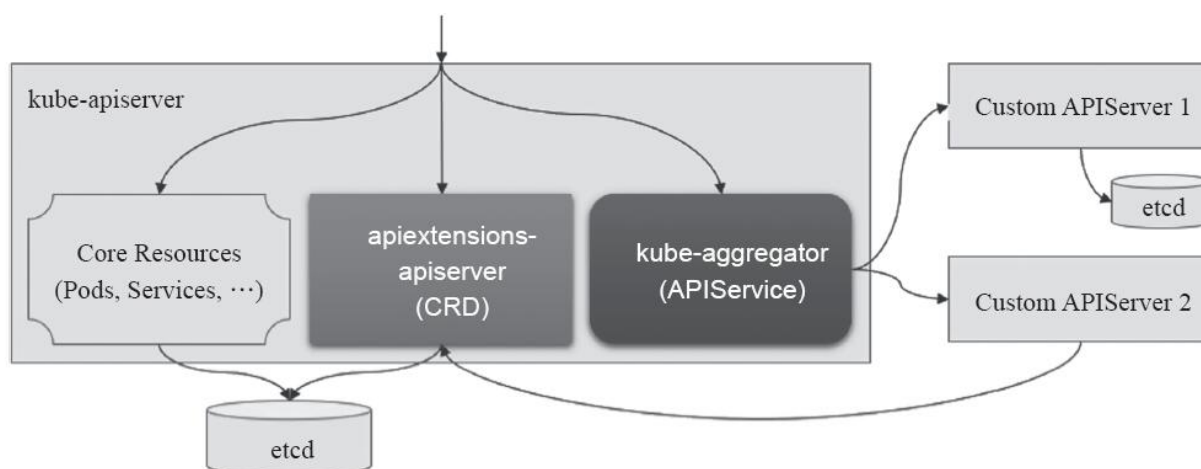


图13-4 自定义API服务器及APIService资源

在使用自定义API Server中的扩展资源之前，管理员需要在主API Server上添加一个相应的APIService资源对象，将自定义API Server注册到kube-aggregator之上，以完成聚合操作。一个特定的APIService对象可用于在主API Server上注册一个URL路径，如/apis/auth.ilinux.io/v1beta1/，从而使得kube-aggregator将发往这个路径的请求代理至相应的自定义API Server。每个APIService对应于一个API群组版本，不同版本的单个API可以由不同的APIService对象支持，如图13-5所示。

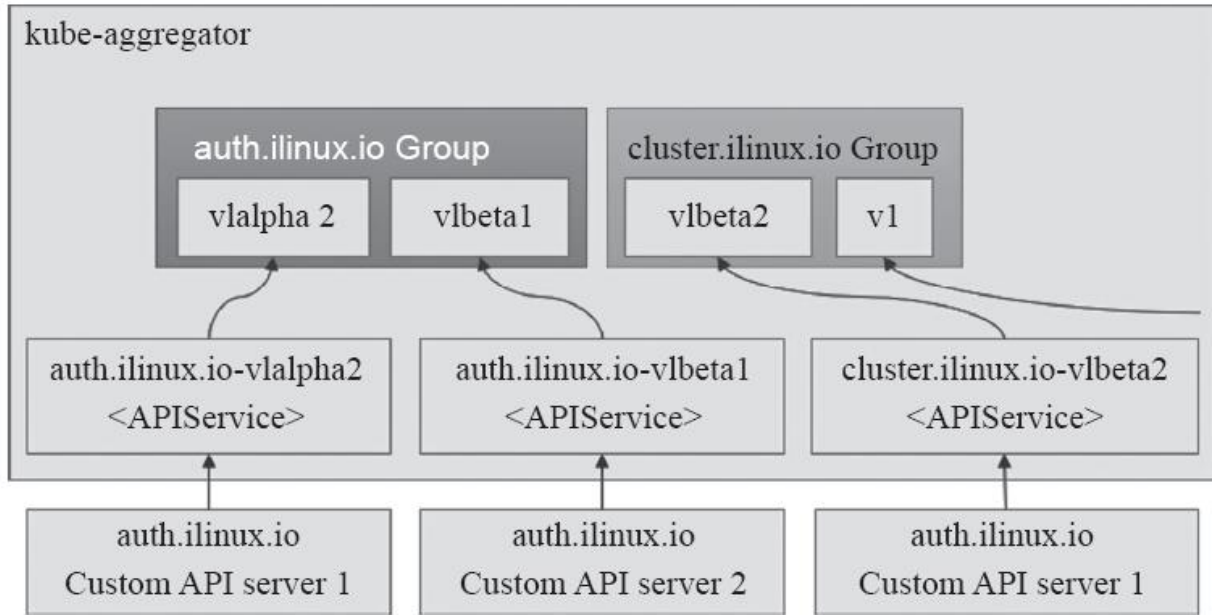


图13-5 Kubernetes聚合层及其聚合方式

相对于CRD和自定义控制器来说，开发自定义API Server要复杂得多，Kubernetes为此也提供了专用的构建聚合API Server的通用库，项目名称就叫作apiserver。apiserver开发库包含了用于创建Kubernetes聚合服务器的基础代码，其中包含委派的authentication和authorization，以及kubectl兼容的发现机制、可选的许可控制链（admission chain）和版本化类型（versioned type）等，同时，它还有一个示例性的项目的sample-apiserver可用作开发模板。不过，目前最好的实践方式是借助于专用于构建聚合服务器的项目apiserver-builder进行开发，它提供了基于apiserver代码构建原生Kubernetes扩展资源的开发库和开发工具的集合。

但创建自定义API Server需要编写大量的代码，而且每个自定义的API Server都需要自行管理所使用的存储系统，它们可使用自有的etcd存储服务，也可通过CRD将资源数据存储在主API Server的存储系统中，但此场景需要事先创建依赖到的所有自定义资源。除非特别需要创建自定义的API Server，否则还是建议读者选择使用Kubebuilder直接基于CRD和自定义控制器进行系统扩展。

13.2.2 APIService对象

APIService资源类型的最初设计目标是用于将庞大的主API Server分解成多个小型但彼此独立的API Server，但它也支持将任何遵循Kubernetes API设计规范的自定义API Server聚合进主API Server中。

下面的配置清单示例取自sample-apiserver项目，它定义了一个名为v2beta1.auth.ilinux.io的APIService对象，用于将default名称空间中名为auth-api的Service对象后端的自定义API Server聚合进主API Server中：

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v2beta1.auth.ilinux.io
spec:
  insecureSkipTLSVerify: true
  group: auth.ilinux.io
  groupPriorityMinimum: 1000
  versionPriority: 15
  service:
    name: auth-api
    namespace: default
  version: v2beta1
```

定义一个APIService对象时，其Spec嵌套使用的字段包括如下这些字段。

·group<string>: 注册使用的API群组名称。

·groupPriorityMinimum<integer>: API群组的最低优先级，较高优先级的群组将优先被客户端使用；数值越大优先级越高，数值相同时则按名称字符进行排序。

·version<string>: 注册的API群组的版本。

·versionPriority<integer>: 当前版本在其所属的API群组内的优先级；必须使用正整数数值，数值越大优先级越高，数值相同时则按名称字符进行排序。

·**service<Object>**: 自定义API Server相关的Service对象，是真正提供API服务的后端，它必须通过443端口进行通信。

·**caBundle<string>**: PEM编码的CA打包信息，用于验证API service的服务证书。

·**insecureSkipTLSVerify<boolean>**: 与此服务通信时是否禁止TLS证书认证。

APIService仅用于将API Server进行聚合，真正提供服务的是相应的外部API Server，这个自定义服务器通常应该以Pod的形式托管运行于当前Kubernetes集群之上。于是，在Kubernetes集群上部署使用自定义API Server主要由两步组成，首先需要将自定义API Server以Pod形式运行于集群之上并为其创建Service对象，而后创建一个专用的APIService对象与主API Server完成聚合。Kubernetes系统的系统资源指标API就由metrics-server项目提供，该项目基于metrics-server提供了一个扩展的API，并通过API群组metrics.k8s.io将其完成聚合，后文中资源指标及监控相关的章节将讲述相关的部署及使用方法。

13.3 Kubernetes集群高可用

Kubernetes具有自愈能力，当它跟踪到某工作节点发生故障时，控制平面可以将离线节点上的Pod对象重新编排至其他可用的工作节点上运行，因此，更多的工作节点也就意味着更好的容错能力，因为它使得Kubernetes在实现工作节点故障转移时拥有更加灵活的自由度。而当管理员检测到集群负载过重或无法容纳其更多的Pod对象时，通常需要手动将节点添加到集群，其过程略为烦琐，Kubernetes cluster-autoscaler还为集群提供了规模按需自动缩放的能力。

然而，添加更多的工作节点并不能使集群适应各种故障，例如，若主API服务器出现故障（由于其主机出现故障或网络分区将其从集群中隔离），则其将无法再跟踪和控制集群。因此，还需要冗余控制平面的各组件以实现主节点的服务高可用性。基于冗余数量的不同，控制平面能容忍一个甚至是多个节点的故障。一般来说，高可用控制平面至少需要三个Master节点来承受最多一个Master节点的丢失，才能保证等待状态的Master节点能够保持半数以上，以满足节点选举时的法定票数。一个最小化的Master节点高可用架构如图13-6所示。

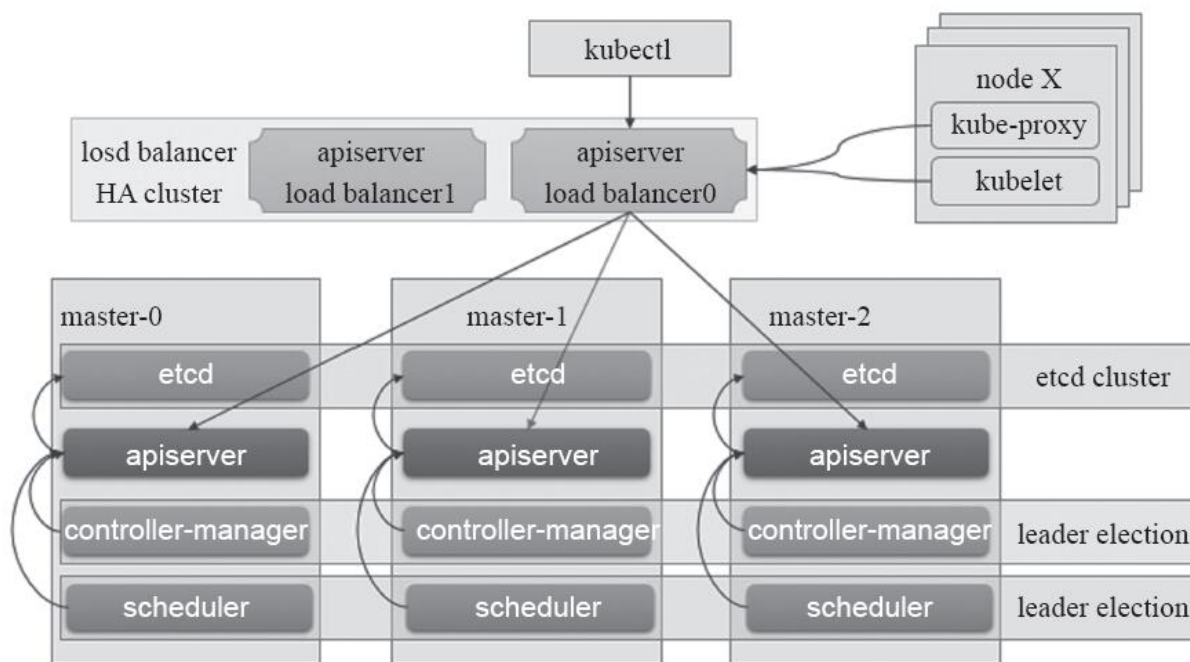


图13-6 最小化Master节点高可用架构

Kubernetes组件中仅etcd需要复杂逻辑完成集群功能，其他组件间的松耦合特性使得系统能够通过多种方式实现Master节点的高可用性，图13-6是较为常用的一种架构，各架构方式也通常有一些共同的指导方针，具体如下。

- 利用etcd自身提供的分布式存储集群为Kubernetes构建一个可靠的存储层。

- 将无状态的apiserver运行为多副本，并在其前端使用负载均衡器调度请求；需要注意的是，负载均衡器本身也需要是高可用的。

- 多副本的控制器管理器，通过其自带的leader选举功能（--leader-election）选举出主角色，余下的副本在主角色发生故障时自动启动新一轮的选举操作。

- 多副本的调度器，通过其自带的leader选举功能（--leader-election）选举出主角色，余下的副本在主角色发生故障时自动启动新一轮的选举操作。

13.3.1 etcd高可用

分布式服务之间进行可靠、高效协作的关键前提是有一个可信的数据存储和共享机制，**etcd**项目正是致力于此目的构建的分布式数据存储系统，它以键值格式组织数据，主要用于配置共享和服务发现，也支持实现分布式锁、集群监控和**leader**选举等功能。

etcd基于Go语言开发，内部采用**raft**协议作为共识算法进行分布式协作，将数据同步存储在多个独立的服务实例上以提高数据的可靠性，从而避免了单点故障所导致的数据丢失。**Raft**协议通过选举出的**leader**节点来实现数据的一致性，由**leader**节点负责所有的写入请求并同步给集群中的所有节点，在取决半数以上**follower**节点的确认后予以持久存储。这种需要半数以上节点投票的机制要求集群数量最好是奇数个节点，推荐的数量为3个、5个或7个。**etcd**集群的建立有三种方式，具体如下。

- 静态集群：事先规划并提供所有节点的固定IP地址以组建集群，仅适合于能够为节点分配静态IP地址的网络环境，好处是它不依赖于任何外部服务。

- 基于**etcd**发现服务构建集群：通过一个事先存在的**etcd**集群进行服务发现来组建新集群，支持集群的动态构建，它依赖于一个现存可用的**etcd**服务。

- 基于DNS的服务资源记录构建集群：通过在DNS服务上的某域名下为每个节点创建一条SRV记录，而后基于此域名进行服务发现来动态组建新集群，它依赖于DNS服务及事先管理妥当的资源记录。

一般说来，对于**etcd**分布式存储集群来说，三节点集群可容错一个节点，五节点集群可容错两个节点，七节点集群可容错三个节点，依次类推，但通常来说，多于七个节点的集群规模是没有必要的，而且对系统性能也会产生负面影响。

13.3.2 Controller Manager和Scheduler高可用

Controller Manager通过监控API Server上的资源状态变动并按需分别执行相应的操作，于是，多实例运行的kube-controller-manager进程可能会导致同一操作行为被每一个实例分别执行一次，例如，某一Pod对象创建的请求被3个控制器实例分别执行一次进而创建一个Pod对象副本来。因此，在某一时刻，仅能有一个kube-controller-manager实例处于正常工作状态，余下的均处于备用状态，或者称为等待状态。

多个kube-controller-manager实例要同时启用“--leader-elect=true”选项以自动实现leader选举，选举过程完成后，仅leader实例处于活动状态，余下的其他实例均转入等待模式，它们会在探测到leader故障时进行新一轮的选举。与etcd集群基于raft协议进行leader选举不同的是，kube-controller-manager集群各自的选举操作仅是通过在kube-system名称空间中创建一个与程序同名的Endpoints资源对象来实现：

```
~]$ kubectl get endpoints -n kube-system
```

NAME	ENDPOINTS	AGE
kube-controller-manager	<none>	13h
kube-scheduler	<none>	13h
...		

这种leader选举操作是分布式锁机制的一种应用，它通过创建和维护Kubernetes资源对象来维护锁状态，目前Kubernetes支持ConfigMap和Endpoints两种类型的资源锁。初始状态时，各kube-controller-manager实例通过竞争的方式去抢占指定的Endpoints资源锁。胜利者将成为leader，它通过更新相应的Endpoints资源的注解control-plane.alpha.kubernetes.io/leader中的“holderIdentity”为其节点名称，从而将自己设置为锁的持有者，并基于周期性更新同一注解中的“renewTime”以声明自己对锁资源的持有状态从而避免等待状态的实例进行争抢。于是，一旦某leader不再更新renewTime了，等待状态各实例就将一哄而上进行新一轮的竞争。

```
~]$ kubectl describe endpoints kube-controller-manager -n kube-system
```

Name:	kube-controller-manager
Namespace:	kube-system

```

Labels:      <none>
Annotations: control-plane.alpha.kubernetes.io/leader=
{"holderIdentity":"master1.
  ilinux.io_846a3ce4-b0b2-11e8-9a23-
00505628fa03","leaseDurationSeconds":15,"acquireTime":
  "2018-09-05T02:22:54Z","renewTime":"2018-09-
05T02:40:55Z","leaderTransitions":1}'
Subsets:
Events:
  Type      Reason             Age   From              Message
  ----      -
  Normal    LeaderElection    13h   kube-controller-manager
master0.ilinux.io_e8fca6fc-
b049-11e8-a247-000c29ab0f5b became leader
  Normal    LeaderElection    5m    kube-controller-manager master1.ilinux.io_846-
a3ce4-b0b2-11e8-9a23-00505628fa03 became leader

```

kube-scheduler的实现方式与此类似，只不过它使用的是自己专用的Endpoints资源kube-scheduler。



提示 基于kubeadm部署高可用集群的方式可参考文档<https://kubernetes.io/docs/setup/independent/high-availability/> 给出的步骤来进行。

13.4 Kubernetes的部署模式

Kubernetes在容器生态系统的快速增长过程中形成了多种部署模式，包括自助式、自托管式和完全自动化等多种管理形式的集群。无论部署方式如何，开发人员和运营团队都要遵循标准化、一致的工作流程来管理容器化应用程序的生命周期，这也恰是**Kubernetes**的关键优势之一。

实践中，**Kubernetes**的客户可以使用各种各样的部署模型，包括对开发人员友好的**PaaS**模型以及高度自定义的运行于裸服务器的部署模型，等等，其中的每种模式都有其优缺点。通常，将容器部署到本地服务器时，数据的持久存储是最大的挑战，而对于那些部署到云环境的模型来说，引用监视和日志记录则是其最大的挑战。另外，安全性是任何**Kubernetes**部署模型的核心因素，任何部署操作从设计时就应该充分考虑到其安全性。

本节主要描述**Kubernetes**系统较为常用的部署模型，以帮助读者理解其部署选项，与每个选项相关的挑战和考虑因素，以及在其中运行生产型工作负载的管理模型。

13.4.1 关键组件

生产可用的部署方案中，除了Kubernetes之外，还有多个对生产集群来说至关重要的组件，如镜像仓库、监控系统、日志系统等，如图13-7所示。

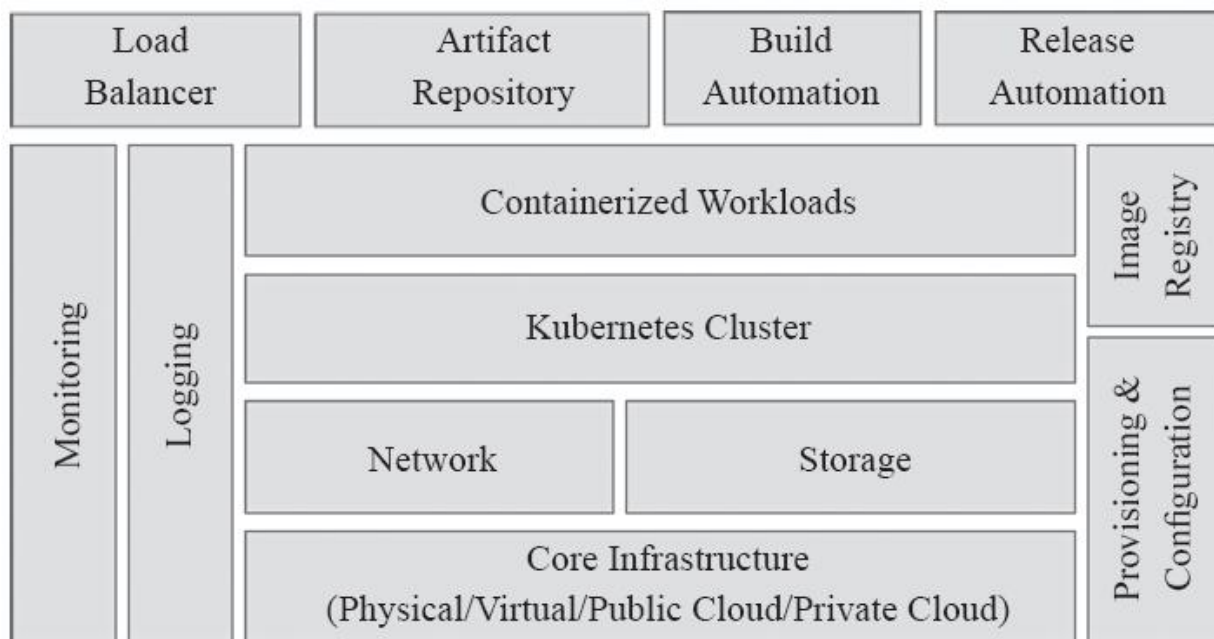


图13-7 Kubernetes集群的关键组件

1) 核心基础架构：此为Kubernetes集群的基础设施组件，用于提供支持容器化工作负载的计算、网络及存储相关的底层部件，它们可能是物理服务器、虚拟化数据中心、IaaS类型的私有云或公有云。

2) 叠加网络：Kubernetes基于SDN（Software-Defined Networking）提供的网络环境实现内部通信，它将用于确保集群的所有内部组件都能够正常通信，可以选用的项目包含诸如Calico、Flannel、Romana和Weave Net等，这在前面的章节中已经有过详细的讲解。

3) 存储系统：运行有状态的工作负载（如数据库）时，持久化数据存储是必备的组件。SDS（Software-Defined Storage）层即作为持久存储卷暴露给容器，分布式存储软件如GlusterFS、网络文件系统（NFS）和块级别存储卷（RBD等）是首选。

4) **Kubernetes**集群：由控制平面**Master**节点、分布式键值存储系统**etcd**和工作节点组成。主节点负责工作负载的调度和编排，主要由**API**服务器、控制器管理器和调度器组成。它们是**Kubernetes**集群的控制中心，生产环境中需要冗余化配置以确保服务的可用性。**etcd**存储系统负责维护集群和工作负载的当前状态，鉴于其重要性通常需要为其配置分布式环境以实现冗余和高可用性。各工作节点负责运行工作负载，在云计算环境中此部分可由**Cluster AutoScaler**实现弹性伸缩。

5) 容器化工作负载：于**Kubernetes**集群内部署的应用程序，其中一部分有可能需要暴露给集群外部的客户端程序。

6) 供应和配置管理：安装和配置**Kubernetes**集群与部署高度可用的关键型分布式应用程序并无太大区别。为了确保一致性和可重复性，通常应该依赖于工具链的实现，类似**Ansible**、**Chef**、**Puppet**、**Terraform**和其他自动化工具。这些工具使得升级、修补和维护**Kubernetes**基础架构变得更加容易。

7) 镜像仓库：运行容器化应用程序时，**Kubernetes**节点需要事先从镜像仓库中提取相应的容器镜像。在每次提交代码时自动构建新镜像的环境中，应用程序将自动升级为最新版本的镜像。为了减少延迟并提高安全性，镜像应该存储在托管于集群之上的仓库服务中。

8) 日志记录和监控：分布式应用程序会生成大量日志，**Kubernetes**也不例外。集群中的每个组件（包括已部署的应用程序）都会生成需要捕获和处理的日志，它们对故障排查和监视群集活动有着不可替代的作用。日志与监控工具结合使用，有助于深入了解集群的运行状态，常用的实现有如**Elastic Stack**、**Grafana**和**Prometheus**等。该层是生产部署的重要组成部分。

9) 负载均衡器：**Kubernetes**集群有两处位置依赖于负载均衡器，它们分别是**API**服务器和公共类型的容器化应用。配置了高可用服务的**Master**节点上，**API**服务器经负载均衡器调度以承载更多的用户请求，而提供公共服务的应用程序运行于多个**Pod**对象并向集群外部暴露时也需要负载均衡器。

10) 工件仓库：工件仓库用于维护属于应用程序的资产，尤其是随着分布式应用程序复杂性的增长，需要管理的程序资产包括各种配

置设定、依赖项、软件包、脚本，甚至是二进制文件。在某些情况下，工件仓库本身甚至也具有镜像仓库的功能。

11) 构建和发布管理：随着持续集成和持续交付成为应用程序生命周期管理（**ALM**）的首选机制，构建和发布自动化正在成为IT组织核心文化，实现此类功能的自动化工具通过高效的流水线来连接源代码管理系统和生产环境。

13.4.2 常见的部署模式

Kubernetes是近来最为成功的开源项目之一，在CNCF的主导下，它得到了来自于CoreOS、Google、华为、IBM、RedHat和中兴通讯等公司程序员的积极贡献，源代码质量较高且经过了社区的严格评估，托管于GitHub之上的代码仓库中的主干代码可直接用于部署生产环境。不过，据CNCF于2017年秋季的调查显示，实施的复杂性仍是许多组织不使用Kubernetes的主要原因之一。

幸运的是，Kubernetes项目在日渐成熟的过程中，社区为系统的安装简化也正在倾力而为。尽管该软件的初始版本安装起来依然稍显复杂，但借助于kubeadm之类的工具已然能够使得普通的系统管理员也可以较为轻松地部署Kubernetes，本书使用的也正是这种部署方式。另外，诸如Cloud Foundry Container Runtime、Canonical conjure-up和kops等工具的可用性也使得Kubernetes在数据中心和公有云环境中的部署变得更加简单。

于是，自定义或自主托管Kubernetes部署的模式正在受到越来越多人的关注，基于此种方式，用户可以在物理服务器、虚拟机和云计算环境中进行系统部署，其配置方式和部署体验据所用工具的不同而有所不同。

自定义的部署模式为用户的终极部署机制，他们可以从众多的目标部署环境、机器配置、操作系统、存储后端、网络插件和高可用性配置中进行选择。当然，在提供多样性选择和控制的同时，维护集群的责任将全部维系于用户自身。因此，此种模式中，用户需要为生产群集提供整个组件堆栈，包括从底层计算、网络 and 存储资源到镜像仓库等的安装、配置和管理。当然，在公共云环境中，一些资源（如虚拟机和块存储设备）则由IaaS服务商管理。

在操作系统、存储后端和叠加网络方面需要高度自定义的组织通常会选择自定义或自托管部署，毕竟此方法还提供了一些其他部署模式中可能无法提供的高级功能。自定义部署是最廉价的选择，因为客户只需要投资基础架构，用到的几乎所有的工具都是开源的，都可以从社区免费获取，但是组织必须考虑维护基础设施所涉及的人员和支持成本。

而那些希望在数据中心或公有云环境中运行Kubernetes而无须安装或维护集群的用户可选择托管的Kubernetes集群产品，提供托管Kubernetes服务的供应商则向用户收取集群的管理和维护费用，为此用户将不得不花费核心基础设施以及订购获得许可费用。托管的Kubernetes平台提供了两全其美的功能：基础设施的选择与免维护集群的结合。不具备安装、配置和管理大规模部署所需技能的组织可以选择托管的Kubernetes平台，如图13-8所示的带有独立边框的组件部分将由服务商负责。

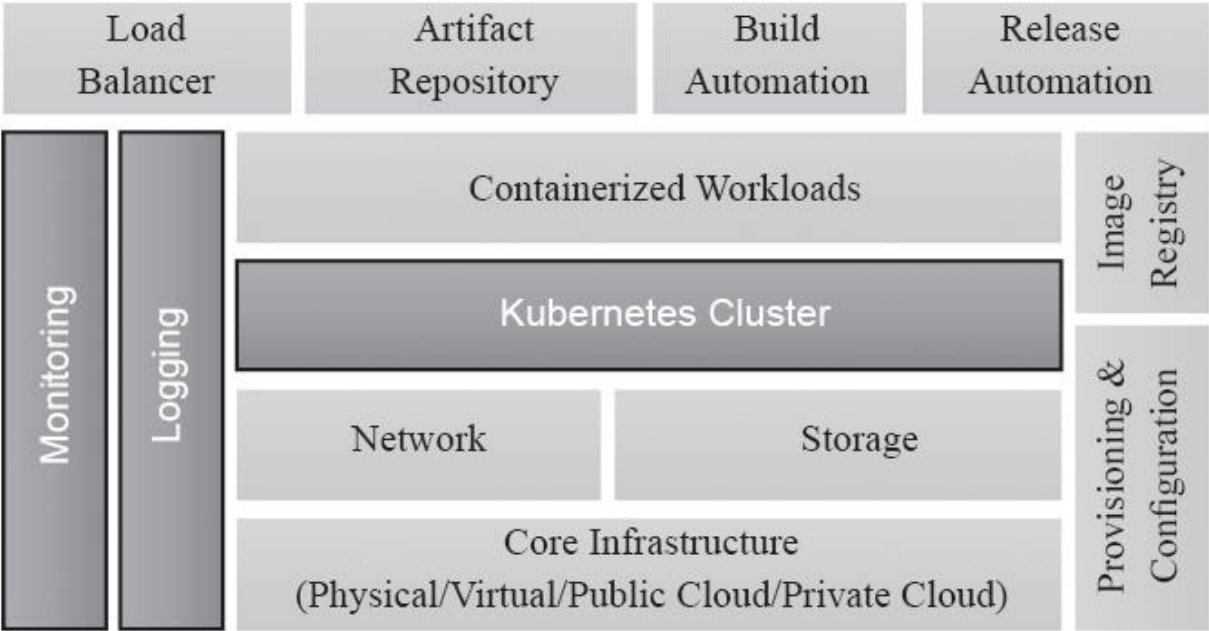


图13-8 托管的Kubernetes集群环境

由于平台供应商可以远程管理集群，因此用户可以专注于应用程序的开发，而不是维护容器基础架构。不过，与自托管部署相比，托管式Kubernetes产品更昂贵，毕竟组织将不得不考虑基础架构成本以及集群管理成本，好在管理Kubernetes平台的定期升级、修补、安全和监控服务的优势可以在长期预算范围内抵消一部分成本。此类的常见解决方案有IBM Cloud Private、Platform9和Tectonic等。

另外，早期的PaaS实现通常是基于隔离应用程序上下文的专有技术，而当Docker成为开源集装箱化技术时，PaaS服务商借助于容器取代了专有的执行环境。如今，大多数的PaaS产品都在容器上进行构建。虽然打包成为容器的程序代码仍然运行在虚拟机或物理服务器上，但Kubernetes已成为管理容器的事实上的编排引擎，那些传统的

PaaS完全可以切换为构建在Kubernetes的基础上，提供端到端的应用程序的生命周期管理服务。

PaaS可以部署在公有云环境中或企业数据中心内部，很多大型组织正在采用PaaS来运行内部应用程序以及面向客户的应用程序，他们希望开发团队获得一致的体验，而不考虑部署目标是私有云还是公共云环境。基于Kubernetes的PaaS产品通过一致的工作流程和部署模式向企业提供了这一保证，甚至，在许多情况下，开发人员甚至不需要知道他们的代码将在Kubernetes集群内运行。这就意味着，PaaS层抽象了Kubernetes的底层细节，并且只公开了开发人员理解的API端点。

事实上，当DevOps工具部署于Kubernetes业务流程之上时，它就扩展成了一个容器类型的PaaS平台。开发人员不必处理打包容器的代码，因为平台包含将源代码转换为镜像的工具，并且平台会处理无状态服务和有状态服务之间的连接，开发人员也无须了解如何配置服务发现机制来发现内部和外部服务，如图13-9所示。不过，尽管PaaS降低了复杂性，但灵活性有所欠缺。

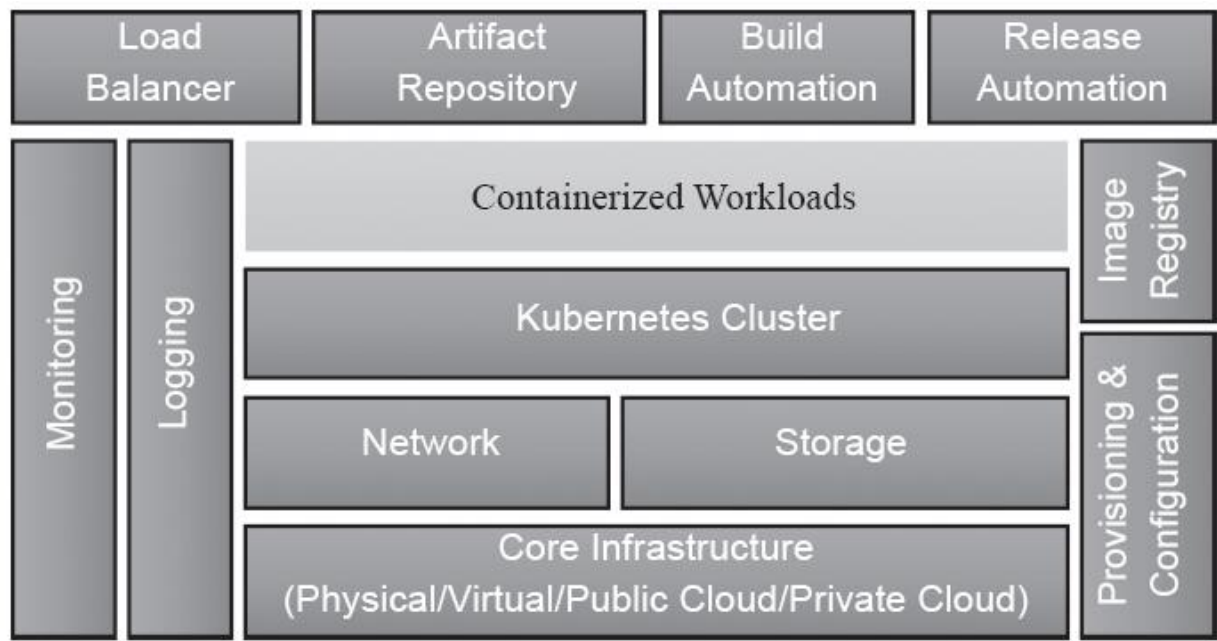


图13-9 基于Kubernetes的PaaS

自托管和托管的Kubernetes集群，其目标是管理员和DevOps团队，而基于Kubernetes的PaaS则是为开发人员所设计的，他们可以将源代码直接应用于平台上，而不是像Docker镜像或Kubernetes Pod那样打包的

工件。他们不需要处理平台相关的任何操作，而仅需要关注代码和应用程序生命周期。Kubernetes已经成为现如今PaaS平台实践的基础，Red Hat OpenShift和Mesosphere DC/OS是著名的基于Kubernetes的PaaS产品。

最后，随着Kubernetes的成熟，其用例开始超越普通的容器编排服务而逐渐被用于多种小众场景中，这包括边缘计算（Edge Computing）、机器学习、无服务计算（Serverless Computing）和数据流分析（Stream Analytics）等，尤其值得一提的是新近兴起的无服务计算。目前，无服务计算与虚拟机和容器一起已成为公有云提供商提供的基本计算服务。不过，与其他计算服务不同的是，无服务计算基于事件驱动，开发人员以功能的形式将代码片段上传到无服务器平台后，它们只会在外部事件触发时执行，而非一直以后台进程处于守护运行状态。AWS Lambda、Azure函数和Google Cloud Functions是公有云中一些流行的无服务器计算选择。Docker可用于无服务器计算，于是Kubernetes成为在运行时管理这些Docker容器的首选项。Apache OpenWhisk、Fission、Kubeless、nuclio和OpenFaaS是基于Kubernetes的部署Serverless的项目。

13.5 容器时代的DevOps概述

DevOps的理论和实践起源于2009年前后，但最初应用的大趋势不温不火，直到容器技术出现并迅速流行开来之后，DevOps才日渐红火起来。究其原因，其中的一个可能性便是，容器是一个催化剂，它让DevOps文化的落地变得更加易于实现。

应用程序的设计架构从单体架构发展到分层架构之后，各种分布式协作的组件通常会由不同的开发团队来维护和交付，这些应用组件可能使用了不同的编程语言，依赖于不同的开发库环境及外部协作关系，这种系统架构部署和维护的复杂度为运维工作带来了极大的挑战，这一点通过13.5节中描述的DevOps各环节层出不穷的可用工具也可见一斑。如今，应用程序的分层架构进一步发展至微服务架构，开发人员获得便利的同时，运维工作的挑战性却有着进一步上升之势。

容器技术的出现使得运维工作复杂度不断上升的难题迅速得以冰消瓦解，DevOps在技术层面的落地不再是眼花缭乱的局面，它们被统一在容器镜像制作、分发和部署的纬度上。通过使用Dockerfile构建容器镜像，应用的配置和部署工作被提前到了编译时进行，这一点改变了之前制作好应用发布于不同的环境中之后再去手动调整环境变量的传统做法，于是，由于系统环境的差异所造成问题出现的可能性几乎降到了最低。

13.5.1 容器：DevOps协作的基础

容器技术解决了不同系统环境上应用程序的配置和维护的复杂度难题，从而使得开发人员和IT运维人员能够更紧密和更高效地协作，为DevOps的快速落地提供了一个极具效用的突破口。有了容器，传统DevOps实践中不得不为各种环境中应用程序的构建和释出进行复杂配置的局面，被容器技术极大地简化了。

借助Docker容器，开发人员可以做到将精力集中于容器中内容的构建（应用程序和服务，以及它们所依赖的框架和组件）以及如何将容器和服务协同在一起工作之上。当然，容器协调的编排机制有赖于编排工具的介入。相对应地，IT运维人员则可以将主要精力集中于生产环境管理方面，例如基础架构、系统扩缩容、监控，以及将应用程序正确交付到终端用户等，他们根本无须关心容器内容本身。

如图13-10所示，开发人员负责开发编写容器中运行的应用程序，他们通过Dockerfile来定义应用程序所依赖的系统环境，以及将代码构建到容器镜像中的步骤，并借助于编排工具的编排机制定义容器的协同方式。遵循业务环境需求，将制作好的Docker配置文件推送至项目的代码仓库中（如Git仓库）即可进入后续的步骤。

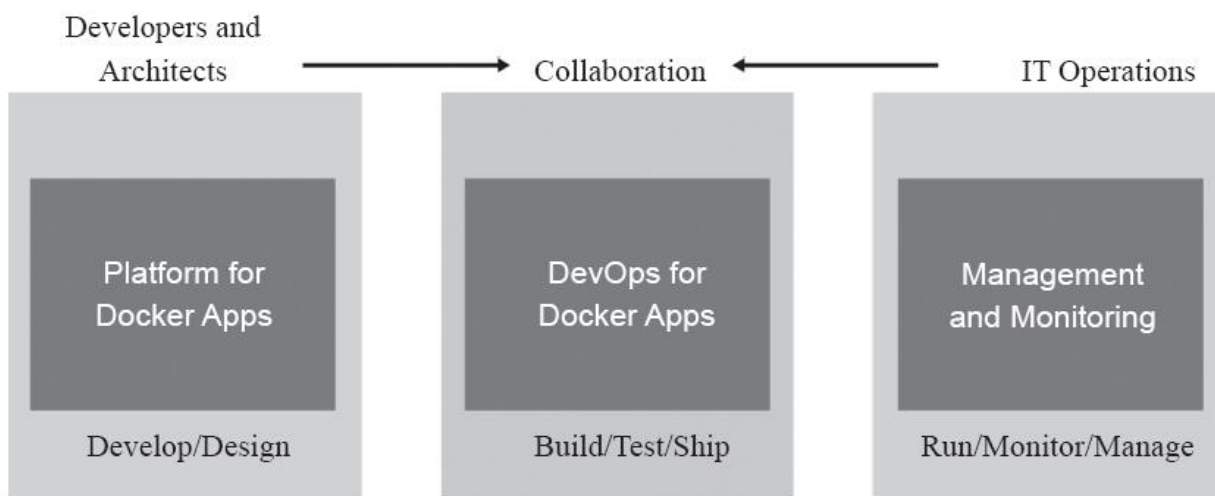


图13-10 容器模型中的DevOps（来源：microsoft.com）

接下来，在DevOps环节持续集成（CI）流水线从Git仓库中获取到的Dockerfile，结合从镜像仓库中拿到基础镜像自动构建自定义的容器镜像，并于制作完成后将其推送至选定的镜像仓库中以用于后续的部署操作。

随后，由CD（持续交付和技术部署）流水线自动完成基础设施构建、环境设定、容器部署和监控等工作，并将系统状态数据不断地反馈至开发团队。对于此环节来说，不同的团队的不同项目的自动化程度可能会有所不同，有的甚至还可能会由运维团队手工完成。

由此可见，开发和运维两个团队在容器平台的辅助下得以完成协同，从而大大缩短了软件开发生命周期。开发人员负责维护容器的内容，而运维人员则负责在编排系统的辅助下将镜像运行为一个个的容器。

13.5.2 泛型端到端容器应用程序生命周期 workflow

图13-11提供了更详细的Docker应用程序生命周期的workflow，其着重描述了DevOps的活动和资产。

DevOps流程始于开发人员，他们在自有的内部循环中编写并调试程序代码，而后将“稳定”状态的代码推送至代码仓库（如Git）中。提交完成后，代码仓库将触发CI和工作流的其余部分。

DevOps workflow不仅仅是一个工具集合，它更是一种软件开发演进而来的文化，它是使得软件开发更敏捷及可预测的人员、流程和相应的工具的集成。于是，采用了容器化工作流的组织就需要基于容器 workflow来重构他们的组织及其工作机制。实践DevOps，以自动化取代手工劳动从而大大提升效率并降低出错的可能性，从而帮助组织或企业内容更高效地协作以获得更好的竞争优势。另外，组织还可以结合云计算按需为系统配置资源，实现更高效的环境管理和资源利用最大化，节约运行成本。

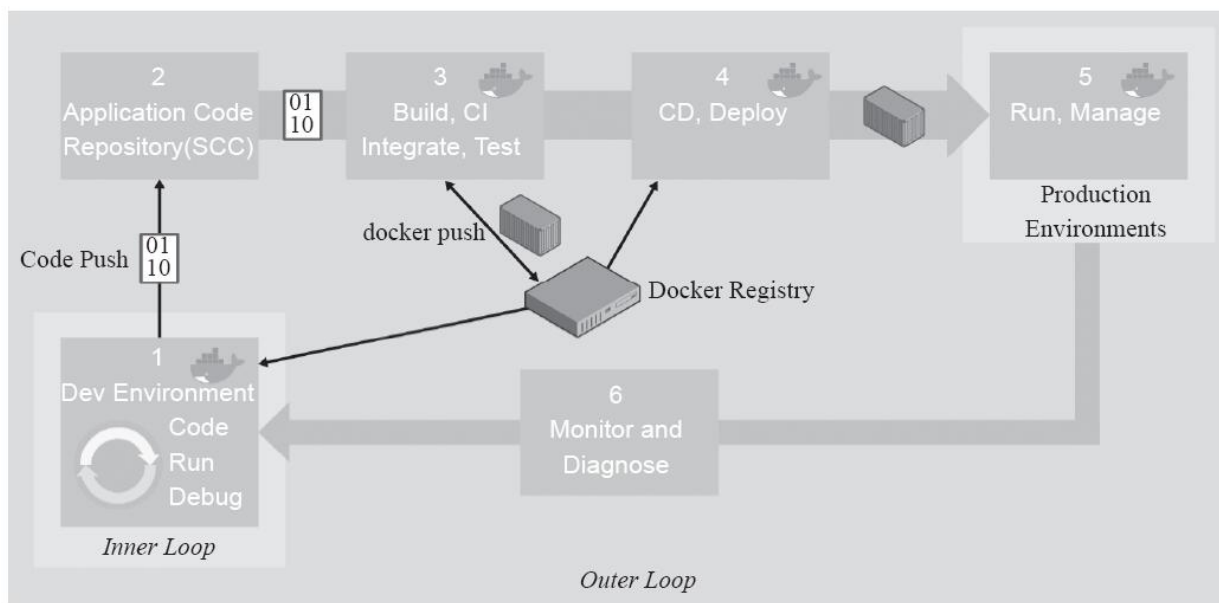


图13-11 泛型端到端容器生命周期 workflow（来源：microsoft.com）

线的Hub，它经由Git WebHook触发控制着CI、镜像构建、Kubernetes Pod创建和编排等一应操作。

此种场景中，Jenkins承担了太多的工作，CI、镜像构建、基础环境、应用部署均由其负责实施，每当有团队需要做一个新的部署流水线时，就会根据具体需求进行微调，必要时还得重写所有的脚本，效率略低。对于有研发实力的组织，可以把关键的工作做成单独的系统，例如，不再把构建组件和部署组件作为Jenkins的插件，而是分别做成独立的系统。

另外，DevOps的流程中需要的各种工具链几乎都可以托管于Kubernetes平台之上运行，包括Jenkins。此时，再加上Kubernetes平台支持的调度、健康检查、日志、监控、安全、应用服务、弹性扩缩容和服务发现等功能，就是一个完整的容器云平台。不过，有些开源项目本身已经直接集成了这些功能，如OpenShift和Tectonic等。

13.6 本章小结

本章主要讲解了自定义资源类型CRD、自定义资源对象、自定义控制器、自定义API Server及API聚合等Kubernetes API的扩展方式，给出了Kubernetes集群高可用架构中控制平面的实现机制，并说明了生产环境中Kubernetes集群的常见部署方式。

第14章 资源指标及HPA控制器

资源监控系统是容器编排系统必不可少的组件，它为用户提供了快速了解系统资源分配和利用状态的有效途径，同时也是系统编排赖以实现的基础要件。在新一代监控架构体系中，**Kubernetes**将资源指标的规范及其实现分离开来，为用户提供了极大的扩展空间。本章将主要说明如何为**Kubernetes**集群提供资源监控机制并利用资源指标。

14.1 资源监控及资源指标

监控应用程序的当前状态是帮助预测问题并发现生产环境中资源瓶颈的最有效方法之一，但它也是目前几乎所有软件开发组织面临的巨大挑战之一，然而，微服务的日益普及使得日志记录和监控变得更加复杂，因为大量正在通信的应用程序本质上是分布式和多样化的，某个单点故障甚至可以中断整个系统，但识别它却变得越来越困难。

当然，监控只是微服务体系中众多挑战中的一个，此外，处理可用性、性能和应用部署等必然地推动了团队创建或使用编排工具来处理所有服务和主机，这也是Kubernetes这一类的编排系统迅速流行的原因之一，因为它能够处理多台计算机中的容器，并消除了处理分布式处理的复杂性。但这么一来问题又转为了如何有效地监控Kubernetes系统并输出指标数据。

众所周知，基于诸如CPU和内存使用等指标自动缩放工作负载规模的能力是Kubernetes最强大的功能之一。当然，要启用此功能，首先需要一种收集和存储这些指标的方法，曾经，这必然是指Heapster。不过，传统的基于Heapster收集的数据指标进行工作负载缩放的方法仅支持CPU一项指标，要扩展为支持多种指标可能有着不小的麻烦，并且来自于项目的各种贡献者的支持也不一致，因此它不久后可能会被淘汰。幸运的是，Kubernetes新的指标API实现了一种更为一致和高效的供给指标数据的方式，这为以自定义指标为基础的自动缩放目标提供了可行的实现。

14.1.1 资源监控及Heapster

Kubernetes有多个数据指标需要采集相关的数据，而这些指标大体上可以分为两个主要组成部分：监控集群本身和监控**Pod**对象。在集群监控层面，目标是监控整个**Kubernetes**集群的健康状况，包括集群中的所有工作节点是否运行正常、系统资源容量大小、每个工作节点上运行的容器化应用的数量以及整个集群的资源利用率等，它们通常可分为如下一些可衡量的指标。

1) 节点资源状态：这个领域的众多指标都与资源利用状况有关，主要有网络带宽、磁盘空间、**CPU**和内存的利用率；基于这些度量指标，管理员能够评估集群规模的合理性。

2) 节点数量：当今，众多公有云服务商均以客户使用实例数量计算费用，于是即时了解到集群中的可用节点数量可以为用户计算所需要支付的费用提供参考指标。

3) 运行的**Pod**对象：正在运行的**Pod**对象数量能够用于评估可用节点的数量是否足够，以及在节点发生故障时它们是否能够承接整个工作负载。

另一方面，**Pod**资源对象的监控需求大体上可以分为三类：**Kubernetes**指标、容器指标和应用程序指标。

1) **Kubernetes**指标：用于监视特定应用程序相关的**Pod**对象的部署过程、当前副本数量、期望的副本数量、部署过程进展状态、健康状况监测及网络服务器的可用性等，这些指标数据需要经由**Kubernetes**系统接口获取。

2) 容器指标：容器的资源需求、资源限制以及**CPU**、内存、磁盘空间、网络带宽等资源的实际占用状况等。

3) 应用程序指标：应用程序自身内建的指标，通常与其所处理的业务规则相关，例如，关系型数据库应用程序可能会内建用于暴露索引状态有关的指标，以及表和关系的统计信息等。

监控集群所有节点的一种方法是通过DaemonSet控制器在各节点部署一个用于监控指标数据采集功能的Pod对象运行监控代理程序（agent），并在集群上部署一个收集各节点上由代理程序采集的监控数据的中心监控系统，统一进行数据的采集、存储和展示。这种部署方式给了管理员很大的自主空间，但也必定难以形成统一之势。

于是，Kubernetes系统特地于kubelet程序中集成相关的工具程序cAdvisor，用于对节点上的资源及容器进行实时监控及指标数据采集，支持的相关指标包括CPU、内存使用情况、网络吞吐量及文件系统使用情况等，并可通过TCP的4194端口提供一个Web UI（kubeadm的默认部署未启用此功能）。需要快速了解某特定节点上的cAdvisor运行是否正常，以及了解单节点的资源利用状态时，可直接访问cAdvisor Web UI，URL是http://<Node_IP>:4194/，其默认的界面如图14-1所示。

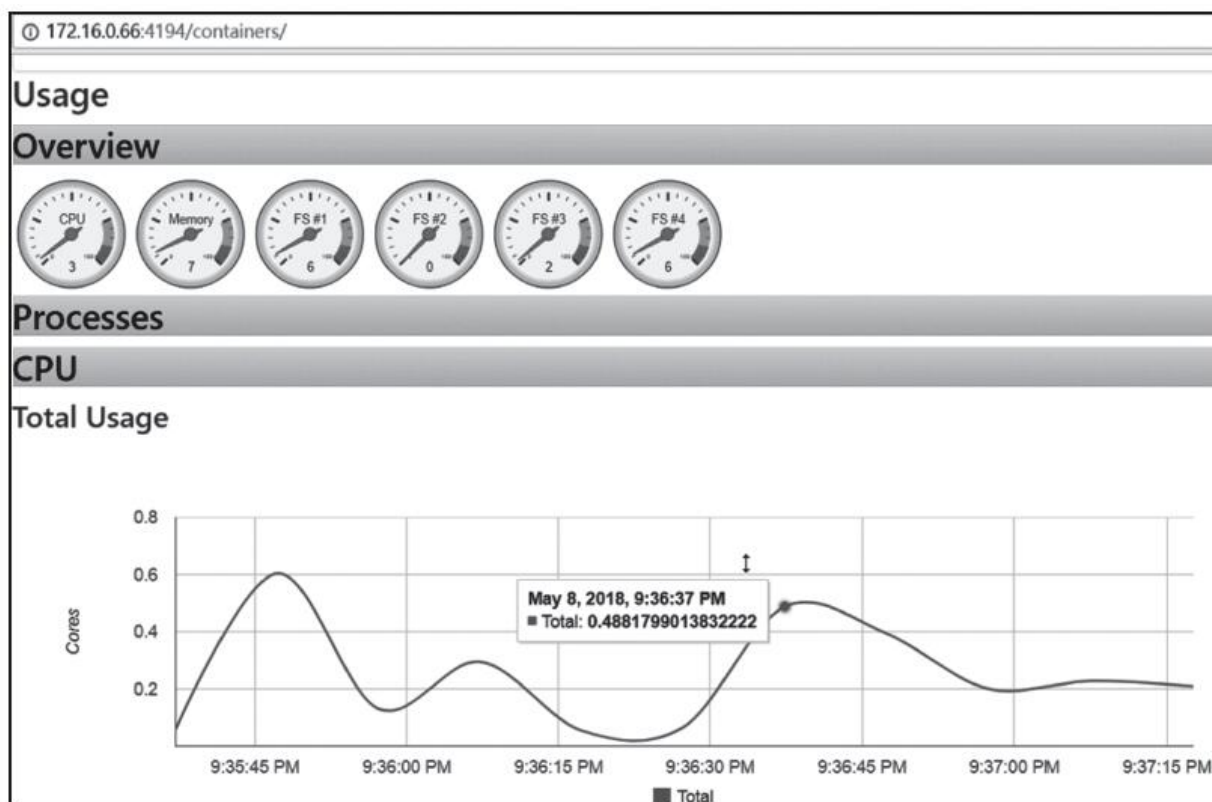


图14-1 cAdvisor图形面板

cAdvisor的问题同样在于其仅能收集单个节点及其相关Pod资源的相关指标数据。事实上，将各工作节点采集的指标数据予以汇集并通过一个统一接口向外暴露不仅能为用户带去便捷，而且也是Kubernetes系统某些组件所依赖的底层功能，这些组件包括kubectl top命令、

Horizontal Pod Autoscalers（即HPA）资源以及Dashboard的某些功能等。

```
~]$ kubectl top nodes
Error from server (NotFound): the server could not find the requested resource
(get services http:heapster:)
```

Heapster是这类项目的一个著名实现，用于为集群提供指标API及其实现，并进行系统监控，而且它曾与ClusterDNS、Ingress和Dashboard一起并称为Kubernetes的四大核心附件，其组件结构如图14-2所示。

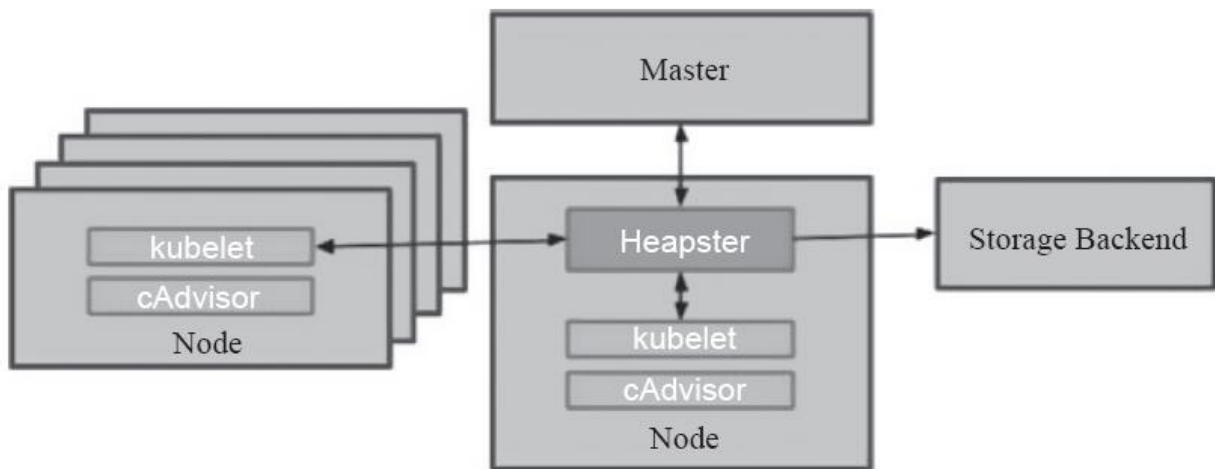


图14-2 Heapster系统数据流向

Heapster是集群级别的监视和事件数据聚合工具，它原生支持并且适用于所有方式构建的Kubernetes集群系统。Heapster本身可作为集群中的一个Pod对象运行，它通过发现集群中的所有节点实现从每个节点kubelet内建的cAdvisor获取性能和指标数据，并通过标签将Pod对象及其相关的监控数据进行分级、聚合后推送到可配置的后端存储系统进行存储和可视化。

功能完备的Heapster监控系统流行的解决方案是由InfluxDB作为存储后端，Grafana为可视化接口，而Heapster从各节点的cAdvisor采集数据并存储于InfluxDB中，由Grafana进行展示。托管于Kubernetes集群中的InfluxDB、Grafana和Heapster运行为常规的Pod资源对象，它们彼此之间通过环境变量及服务发现功能自动协同。另外，它还支持其他多种后端，如OpenTSDB、Elasticsearch、Log、Monasca、Kafka、

Riemann和Hawkular-Metrics等，可按需灵活组合出多种不同的监控解决方案。

然而，Heapster支持的每个存储后端的代码都直接驻留在其代码仓库中成为核心代码库的一部分，结果是必然会被烂尾的驻留代码所拖累，这甚至是成为了用户使用Heapster最常见的挫败原因。更重要的是，即使Heapster没有将Prometheus作为数据接收器，却又暴露了Prometheus格式的指标，这通常会引起不小的混淆和麻烦。

换句话说，Heapster既为那些依赖于指标数据的系统组件提供了指标API（非标准格式的一等类别Kubernetes API），同时又提供了指标API接口背后的实现方式，即收集和存储指标数据以响应对API的请求。这种以耦合度较高的方式实现核心系统组件依赖到的功能部件的做法对系统的变迁引入了不少的不确定性和隐患。

另外，Heapster假设数据存储是一个原始的时间序列数据库，这些数据库都有着一个可直接写入路径。这使得它与Prometheus系统基本上不相兼容，因为Prometheus工作为拉取式模型。然而，Kubernetes生态系统的整体组件几乎原生支持Prometheus系统，于是，Heapster的这部分功能逐渐被其所取代。

为了避免重蹈Heapster的覆辙，资源指标API（resource metrics api）和自定义指标API（custom metrics api）被有意地创建为纯粹的API定义而非具体的实现，它们作为聚合的API安装到Kubernetes集群中，从而允许在API保持不变的情况下切换其具体的实现方案，这一点极大地降低了二者的耦合级别。最终，Heapster用于提供核心指标API的功能也被聚合方式的指标API服务器metrics-server所取代。

14.1.2 新一代监控架构

Kubernetes如此强大的原因之一便是其灵活的可扩展性，其中表现得尤为抢眼的是，它通过API聚合器为开发人员提供了轻松扩展API资源的能力，Kubernetes在1.7版本中引入的自定义指标API（custom metrics API），以及在1.8版本中引入的资源指标API（resource metrics API，简称为指标API）都属于这种类型的扩展。

资源指标API主要供核心系统组件使用，如调度程序、HPA和“`kubectl top`”命令等，虽然是以扩展方式实现API，但它提供的是kubernetes系统必备的“核心指标”，因此不适用于与第三方监控系统集成，如Prometheus等。另一方面，自定义指标API则为用户提供了自行按需扩展指标的接口，它允许用户自定义指标类型的API Server并直接聚合进主API Server中，因此具有更广泛的使用场景。简单总结起来就是，新一代的Kubernetes监控系统架构主要由核心指标流水线和监控指标流水线协同组成，具体如下。

（1）核心指标流水线

由kubelet、资源评估器、metrics-server以及由API Server提供的API群组（由APIService对象提供）组成，如图14-3所示，它们可用于为Kubernetes系统提供核心指标，从而能够了解并操作其内部组件和核心程序。截至目前，相关的指标主要包括CPU累计使用、内存即时使用率、Pod的资源占用率及容器的磁盘占用率等几个。所用到的度量标准核心系统组件包括调度逻辑（基于指标数据的调度程序和应用规模的水平缩放），以及部分UI组件（如`kubectl top`命令和Dashboard）等。

（2）监控指标流水线

监控指标流水线用于从系统收集各种指标数据并提供给终端用户、存储系统以及HPA控制器等使用，它收集的数据指标也称为非核心指标，但它们通常也包含核心指标（未必是Kubernetes可以理解的格式）以及其他指标。自定义指标API允许用户扩展任意数量的特定于应用程序的指标，例如，其指标可能包括队列长度和每秒入口请求数等。Kubernetes系统本身不会提供此类组件，也不会对这些指标提

供相关的解释，它有赖于用户按需选择使用的第三方解决方案。自定义指标API系统组件如图14-4所示。

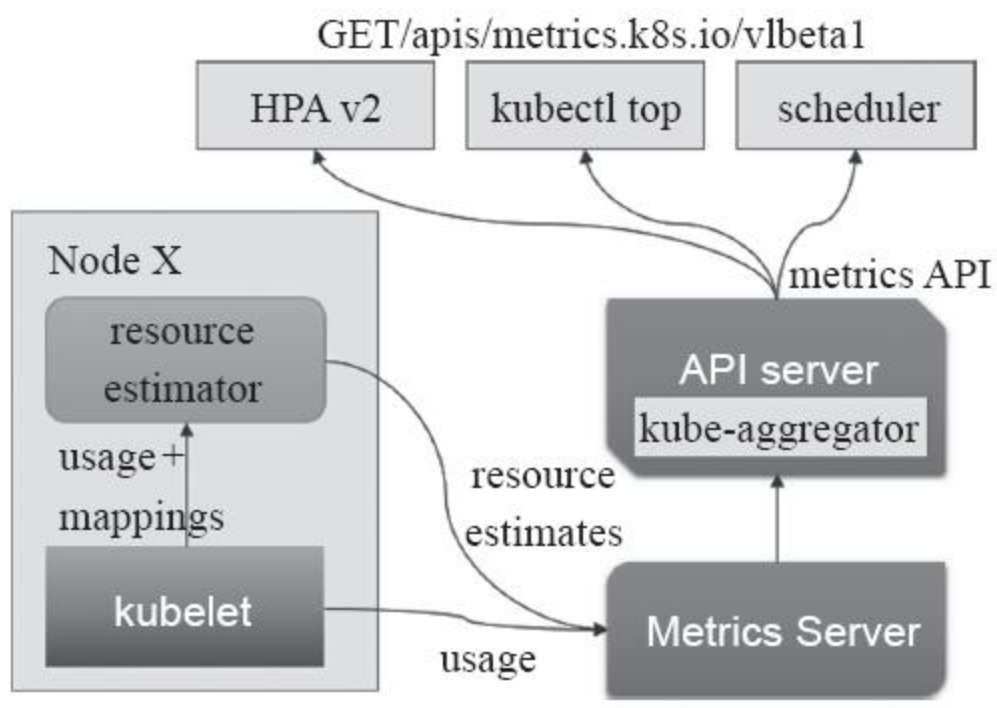


图14-3 资源指标API系统组件

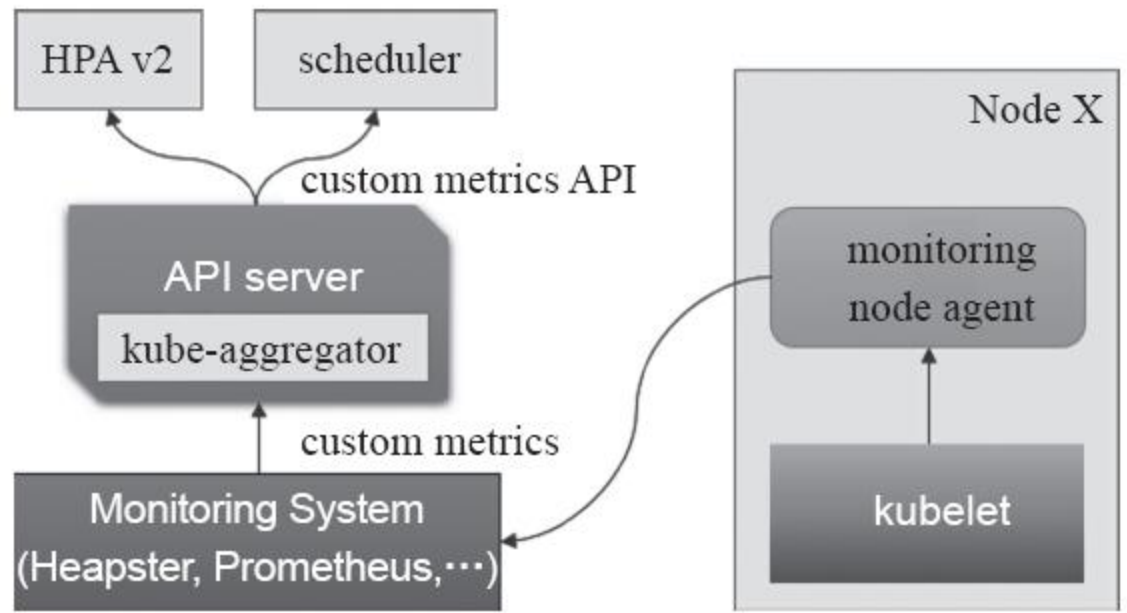


图14-4 自定义指标API系统组件

一个能同时使用资源指标API和自定义指标API的组件是第二版的HorizontalPod-Autoscaler控制器（HPAv2），它实现了基于观察到的指标自动缩放Deployment或ReplicaSet 类型控制器管控下的Pod副本数量。HPA的第一个版本只能根据观察到的CPU利用率进行扩展，尽管在某些情况下很有用，但CPU并不总是最适合自动调整应用程序的度量指标。

从根本上来讲，资源指标API和自定义指标API都仅是API的定义和规范，它们自身都并非具体的API实现。目前，资源指标API的实现较主流的是metrics-server，而自定义指标API则以建构在监控系统Prometheus之上的k8s-prometheus-adapter最广为接受。事实上，Prometheus也是CNCF旗下的项目之一，并且得到了Kubernetes系统上众多组件的原生支持。

14.2 资源指标及其应用

如前所述，自Kubernetes 1.8版本起，诸如容器的CPU和内存资源占用状况一类的资源利用率指标可由客户端通过指标API直接调用，这些客户端包括但不限于终端用户、`kubectl top`命令和HPA（v2）等。另外，尽管通过指标API能够查询某节点或Pod的当前资源占用情况，但API本身并不存储任何指标数据，因此，它仅提供资源占用率的实时监测数据而无法提供过去指定时刻的指标监测记录结果。

14.2.1 部署metrics-server

事实上，资源指标API与系统的其他API并无特别不同之外，它通过API Server的URL路径/apis/metrics.k8s.io/进行存取，并提供同样级别的安全性、稳定性及可靠性保证。不过，只有在Kubernetes集群中部署Metrics Server（指标服务器）应用之后，指标API方才可用。资源指标API架构简图如图14-5所示。

Metrics Server是集群级别的资源利用率数据的聚合器（aggregator），它的创建于不少方面都受到了Heapster的启发，且于功能和特性上完全可被视作一个仅服务于指标数据的简化版的Heapster。Metrics Server通过Kubernetes聚合器（kube-aggregator）注册到主API Server之上，而后基于kubelet的Summary API收集每个节点上的指标数据，并将它们存储于内存中然后以指标API格式提供，如图14-6所示。

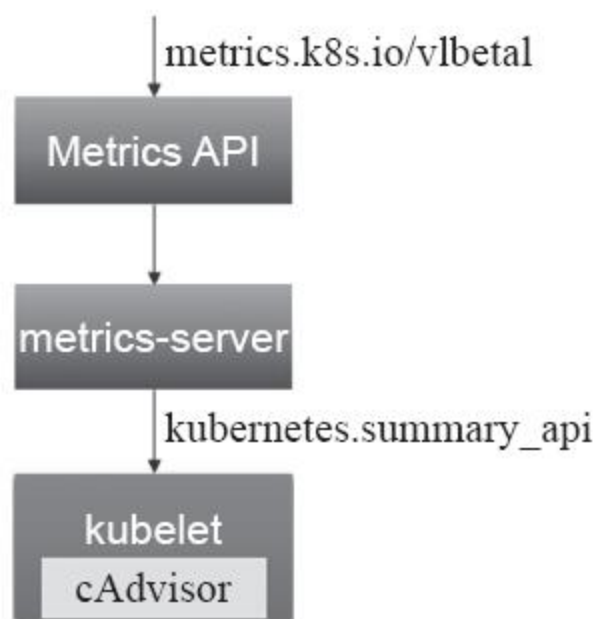


图14-5 资源指标API架构简图

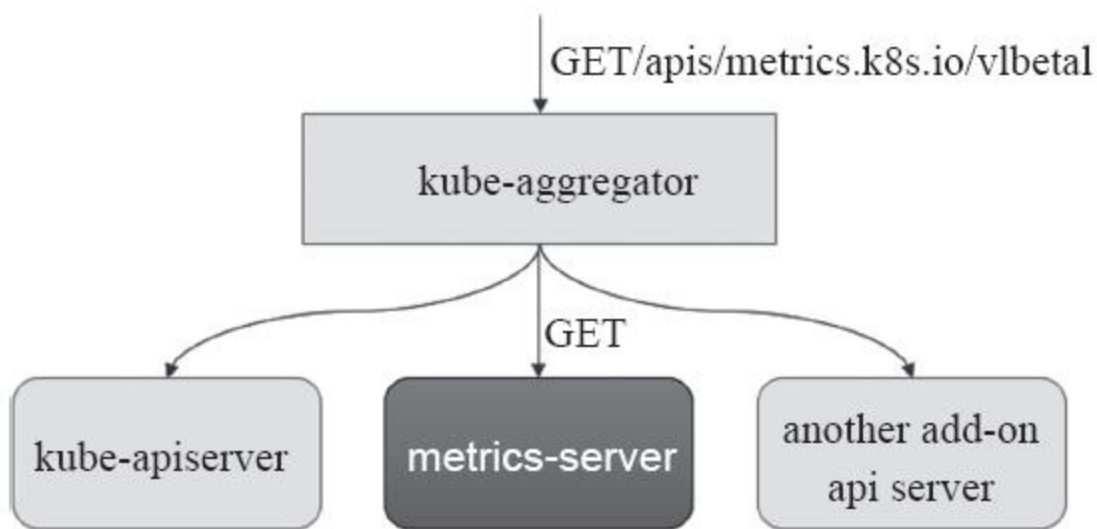


图14-6 聚合metrics-server于主API Server上

Metrics Server基于内存存储，重启后数据将全部丢失，而且它仅能留存最近收集到的指标数据，因此，如果用户期望访问历史数据，就不得不借助于第三方的监控系统（如**Prometheus**等），或者自行开发以实现其功能。

Metrics Server是Kubernetes多个核心组件的基础依赖，因此，它应该默认部署运行于集群中。一般说来，**Metrics Server**在每个集群中仅会运行一个实例，启动时，它将自动初始化与各节点的连接，因此出于安全方面的考虑，它需要运行于普通节点而非**Master**主机之上。直接使用项目本身提供的资源配置清单即能轻松完成**metrics-server**的部署，下面就来分步说明其实施步骤。

(1) 克隆项目代码的仓库至本地目录以获得其资源配置清单

```
~]$git clone https://github.com/kubernetes-incubator/metrics-server.git
```

注意，截至目前，**metrics-server**程序默认会从kublet的基于HTTP通信的10255端口获取指标数据，但出于安全通信的目的，Kubernetes 1.11版本的**kubeadm**在初始化集群时会关掉kublet基于HTTP的10255端口，从而导致其部署完成后无法正常获取数据。另外，代码仓库中的部署清单文件deploy/1.8+/metrics-server-deployment.yaml中并未明确为主程序/metrics-server传递参数指定指标数据的获取接口，它通常应该

是`kubernetes.summary_api`。因此，在启动部署操作之前，需要修改此清单文件，下面在`metrics-server`容器配置段中添加如下内容：

```
command:
- /metrics-server
- --source=kubernetes.summary_api:https://kubernetes.defaultkubeletHttps=true
  &kubeletPort=10250&insecure=true
```

Metrics Server项目处于非常活跃的维护状态，其相应的代码仓库或许在读者读到此书时便已完成了修正，因此建议进行`metrics-server`部署时先不做修改，待部署后观测到连接`kubelet`错误一类的信息时再予以手动更正。

（2）基于资源配置清单完成相应资源的创建

```
~]$ kubectl apply -f metrics-server/deploy/1.8+/

---


```

各部署清单会于`kube-system`名称空间中创建出多个类型的资源对象，包括RBAC相关的`rolebinding`、`clusterrole`和`clusterrolebinding`对象，以及`serviceaccount`对象用于在启用了RBAC授权插件的集群上完成对`metrics-server`开放资源访问的许可。另外，它还会通过一个`APIService`对象创建Metrics API相关的群组从而将`metrics-server`提供的API聚合进主API Server上。因此，接下来要检验相应的API群组`metrics.k8s.io`是否出现在Kubernetes集群的API群组列表中：

```
~]$ kubectl api-versions | grep metrics
metrics.k8s.io/v1beta1
```

（3）确认相关的Pod对象运行正常

首先检查`metrics-server`的Pod对象是否处于“Running”状态：

```
~]$ kubectl get pods -n kube-system -l k8s-app=metrics-server
NAME                                READY    STATUS    RESTARTS   AGE
metrics-server-778ddc45b5-hvwdh    1/1      Running   0           7m
```

而后检查Pod中的容器其日志信息中是否出现错误提示：

```
~]# kubectl logs metrics-server-778ddc45b5-hvwdh -n kube-system
```

(4) 检查资源指标API的可用性

“`kubectl get--raw`”命令可用于基于资源URL路径测试资源指标API服务的可用状态，例如以下命令应返回集群中所有节点的资源使用情况指标列表，它能够列出集群中所有节点的CPU及内存资源占用情况。在使用时，也可以直接给定具体的节点标识，从而仅列出特定节点的相关信息：

```
~]# kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .
{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes"
  },
  .....
}
```



注意 `jq`是专用于处理JSON数据的命令行工具，其功能类似于`sed`对文本信息的处理，在CentOS系统上由`jq`程序包所提供。

此外，**Pod**对象的资源消耗信息也可以经由资源指标API直接列出，例如，要获取集群上所有**Pod**对象的相关资源消耗数据，可使用如下格式的命令：

```
~]# kubectl get --raw "/apis/metrics.k8s.io/v1beta1/pods" | jq
```

资源指标API支持HPA v2控制器根据资源指标（如CPU和内存使用量）进行扩展，但不支持使用特定于应用程序的指标，那些特定于应用程序的指标需要依赖于自定义指标API。执行完成上述步骤并确认结果无误后，**metrics-server**即成功部署完成，那些依赖于核心资源指标的控制器、调度器及UI工具也将在功能上得到进一步的完善。

14.2.2 kubectl top命令

kubectl top命令可显示节点和Pod对象的资源使用信息，它依赖于集群中的资源指标API来收集各项指标数据。它包含有**node**和**pod**两个子命令，可分别用于显示Node对象和Pod对象的相关资源占用率。

列出Node资源占用率命令的语法格式为“**kubectl top node**[-l label[NAME]]”，例如下面显示所有节点的资源占用状况的结果中显示了各节点累计CPU资源占用时长及百分比，以及内容空间占用量及占用比例。必要时，也可以在命令中直接给出要查看的特定节点的标识，以及使用标签选择器进行节点过滤：

```
~]$ kubectl top node
NAME                CPU(cores)   CPU%    MEMORY(bytes)  MEMORY%
master.ilinux.io    444m         11%     1127Mi         65%
node01.ilinux.io    158m         3%      558Mi          32%
.....
```

而名称空间级别的Pod对象资源占用率的使用方式会略有不同，使用时，一般应该限定名称空间及使用标签选择器过滤出目标Pod对象。命令的语法格式为“**kubectl top pod**[-l label][--all-namespaces][--containers=false|true]”，例如，下面显示kube-system名称空间中标签为“k8s-app=kube-dns”的所有Pod资源及其容器的资源占用状态：

```
~]$ kubectl top pod -l k8s-app=kube-dns --containers=true -n kube-system
POD                NAME        CPU(cores)   MEMORY(bytes)
coredns-78fcd6894-ncxdj   coredns     3m           18Mi
coredns-78fcd6894-pvv88   coredns     3m           17Mi
```

kubectl top命令为用户提供了简洁、快速获取Node对象及Pod对象占用系统资源状况的接口，是集群运行和维护的常用命令之一。

14.3 自定义指标与Prometheus

除了资源指标之外，用户或管理员需要了解的指标数据还有很多，如Kubernetes指标、更全面的容器指标、更全面的节点资源指标及应用程序指标，等等。自定义指标API允许请求任意指标，其指标API的实现要特定于相应的后端监视系统。Prometheus是第一个开发了相应适配器的监控系统，毕竟它是监控Kubernetes的第一选择。这个适用于Prometheus的Kubernetes Custom Metrics Adapter由托管在GitHub上的k8s-prometheus-adapter项目提供，如图14-7所示。

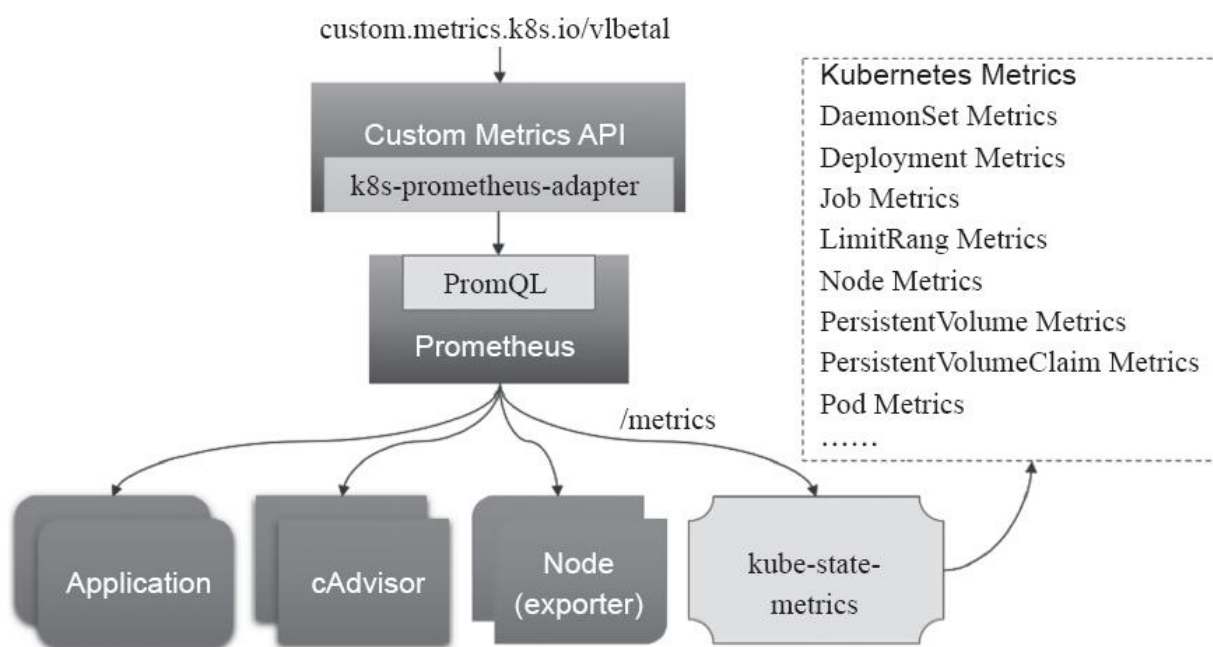


图14-7 自定义指标

自定义指标API的目的是提供最终用户和Kubernetes系统组件可以依赖的稳定的、版本化的API，但其可用的实现及可用指标则有赖于第三方或用户的自行实现，目前基于Prometheus收集和存储指标数据，并借助于k8s-prometheus-adapter将这些指标数据查询接口转换为标准的Kubernetes自定义指标是较为流行的解决方案之一。

14.3.1 Prometheus概述

Prometheus是一个开源的服务监控系统 and 时序数据库，由社交音乐平台SoundCloud在2012年开发，也是CNCf除Kubernetes之外收录的第二款产品，目前已经成为Kubernetes生态圈中的核心监控系统，而且越来越多的项目（如etcd等）都提供了对Prometheus的原生支持，这足以证明社区对它的认可程度。

Prometheus提供了通用的数据模型和便捷的数据采集、存储和查询接口。其核心组件Prometheus服务器定期从静态配置的监控目标（targets）或者基于服务发现（service discovery）自动配置的目标中拉取数据，新拉取到的数据大于配置内存缓存区时，数据将持久化到存储设备中，包括远程云端存储系统。Prometheus的生态组件架构如图14-8所示。

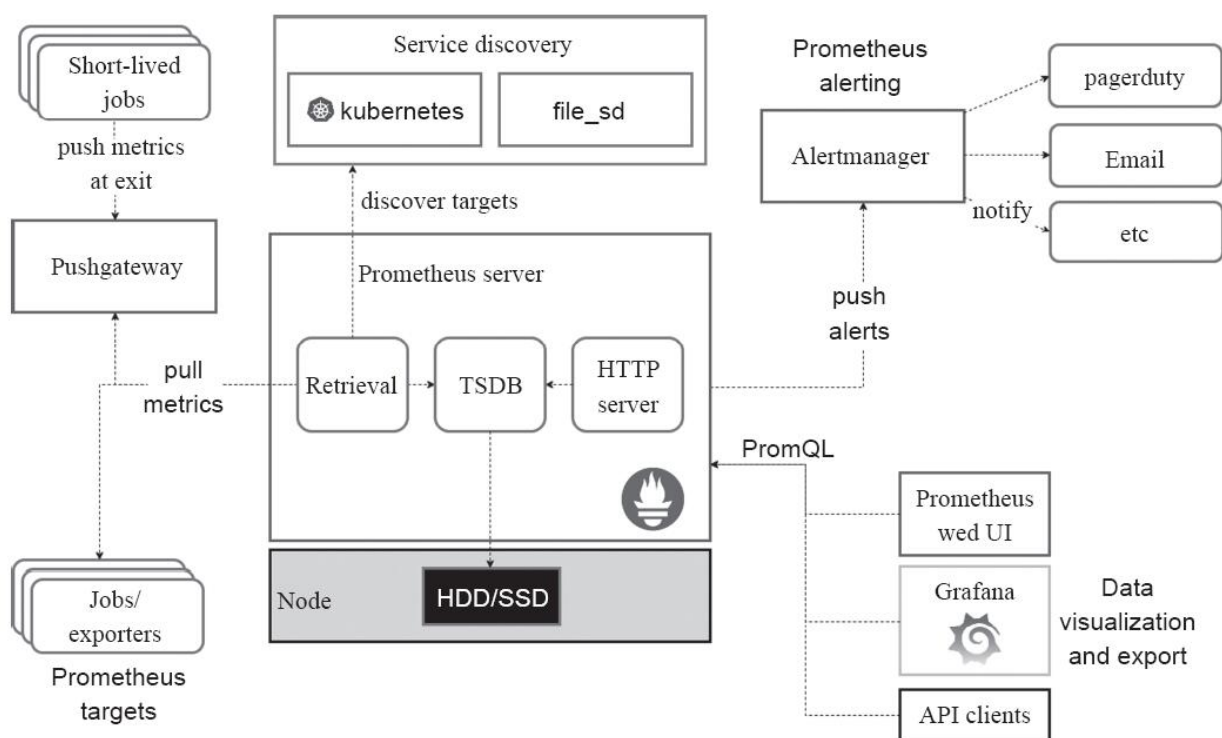


图14-8 Prometheus的生态组件架构

图14-8所示的各种组件中，每个被监控的目标（主机、应用程序等）都可通过专用的exporter程序提供输出监控数据的接口，并等待

Prometheus服务器周期性的数据抓取操作。若存在告警规则，则抓取到数据后会检查并根据规则进行计算，满足告警条件即会生成告警，并发送到Alertmanager完成告警的汇总和分发等操作。被监控目标有主动推送数据的需求时，可部署Pushgateway组件接收并临时存储数据，并等待Prometheus服务器完成数据采集。

任何被监控的目标都需要事先纳入到监控系统中才能进行时序数据采集、存储、告警及相关的展示等，监控目标既可以通过配置信息以静态形式指定，也可以让Prometheus通过服务发现机制动态管理（增删等），对于变动频繁的系统环境（如容器云环境）来说，这种动态管理机制尤为有用。在Kubernetes集群及相关的环境中，除了此前配置的从Pod对象中的容器应用获得资源指标数据以外，Prometheus还支持通过多个监控目标采集Kubernetes监控架构体系中的所谓的“非核心指标数据”，如图14-9所示。

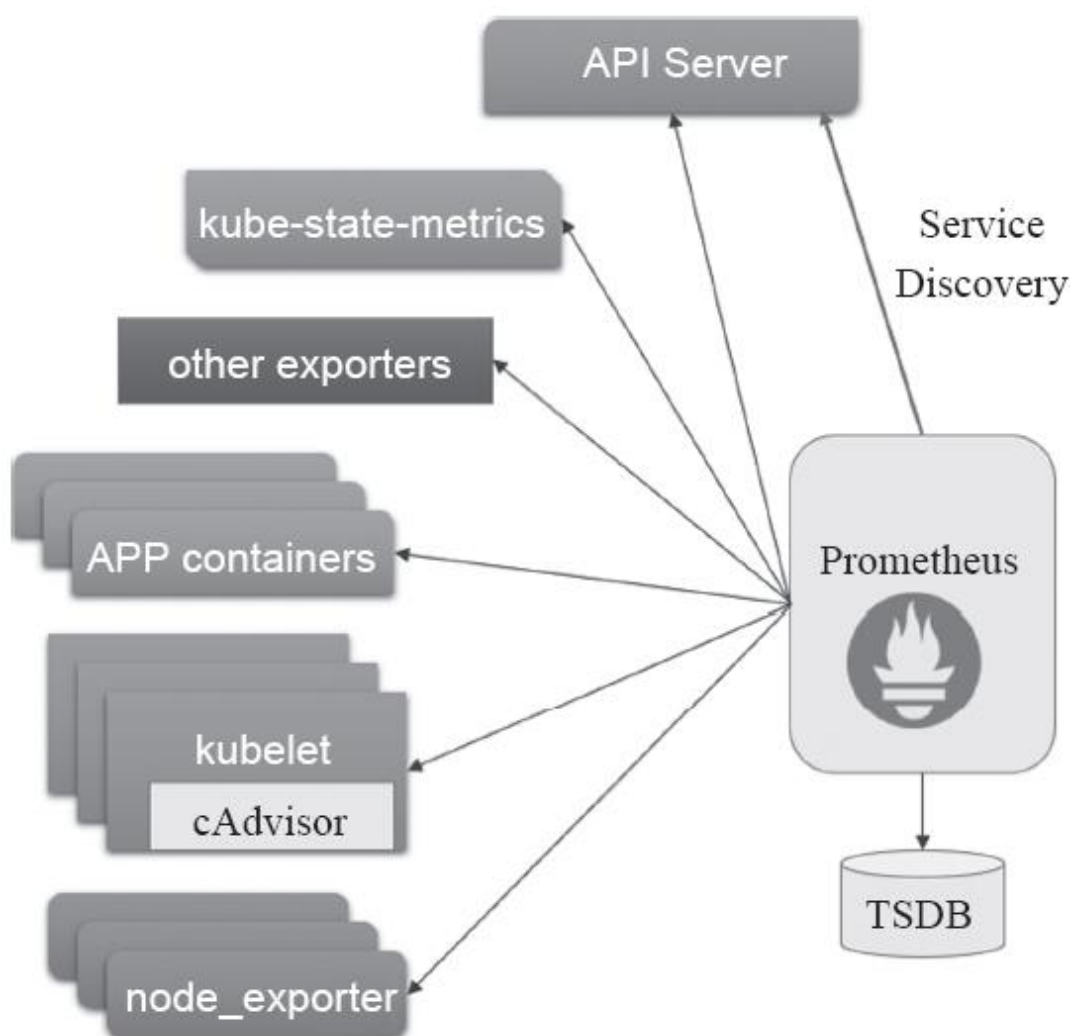


图14-9 Kubernetes中的Prometheus数据源

1) 监控代理程序，如node_exporter，

收集标准的主机指标数据，包括平均负载、CPU、Memory、Disk、Network及诸多其他维度的数据，独立的指标可能多达上千个。

2) kubelet (cAdvisor)：收集容器指标数据，它们也是所谓的Kubernetes“核心指标”，每个容器的相关指标数据主要有CPU利用率（user和system）及限额、文件系统读/写限额、内存利用率及限额、网络报文发送/接收/丢弃速率等。

3) API Server：收集API Server的性能指标数据，包括控制工作队列的性能、请求速率与延迟时长、etcd缓存工作队列及缓存性能、普通进程状态（文件描述符、内存、CPU等）、Golang状态（GC、内存和线程等）。

4) etcd：收集etcd存储集群的相关指标数据，包括领导节点及领域变动速率、提交/应用/挂起/错误的提案次数、磁盘写入性能、网络与gRPC计数器等。

5) kube-state-metrics：此组件能够派生出Kubernetes相关的多个指标数据，主要是资源类型相关的计数器和元数据信息，包括指定类型的对象总数、资源限额、容器状态（ready/restart/running/terminated/waiting）以及Pod资源的标签系列等。

Prometheus能够直接把Kubernetes API Server作为服务发现系统使用进而动态发现和监控集群中的所有可被监控的对象。这里需要特别说明的是，Pod资源需要添加下列注解信息才能被Prometheus系统自动发现并抓取其内建的指标数据。

1) prometheus.io/scrape：用于标识是否需要被采集指标数据，布尔型值，true或false。

2) prometheus.io/path：抓取指标数据时使用的URL路径，一般为/metrics。

3) prometheus.io/port：抓取指标数据时使用的套接字端口，如8080。

另外，仅期望Prometheus为后端生成自定义指标时仅部署Prometheus服务器即可，它甚至也不需要数据持久功能。但若配置完整功能的监控系统，管理员还需要在每个主机上部署node_exporter、按需部署其他特有类型的exporter以及Alertmanager。

14.3.2 部署Prometheus监控系统

为了便于用户快速集成一个完整的Prometheus监控环境，Kubernetes源代码的集群附件目录中统一提供了Prometheus、Alertmanager、node_exporter和kube-state-metrics相关的配置清单，路径为cluster/addons/prometheus，每个项目的配置清单不止一个且文件都以项目名称开起。将Kubernetes源码仓库克隆至本地，以便在后面的各配置步骤中应用其配置文件：

```
~]$git clone https://github.com/kubernetes/kubernetes.git
```

为了便于读者理解和排除问题，下面分步骤分别说明kube-state-metrics、node_exporter、Alertmanager和Prometheus四个组件的部署过程。

1.生成集群资源状态指标

kube-state-metrics能够从Kubernetes API Server上收集各大多数资源对象的状态信息并将其转为指标数据，它工作于HTTP的/metrics接口，通过Prometheus的Go语言客户端暴露于外，Prometheus系统以及任何兼容相关客户端接口的数据抓取工具都可采集相关数据，并予以存储或展示。因此，再结合Prometheus的其他指标数据采集接口及报警功能，用户可于Kubernetes构建出一个自定义指标API Server的同时提供一个完整的监控系统。

kube-state-metrics是Kubernetes集群的一个附加组件，它通过Kubernetes API service监听资源对象的状态并生成相关的指标，不过，它不在意单个Kubernetes组件的运行状况，而是关注内部各种对象的整体运行状况，如Deployment对象或ReplicaSet对象等。换句话说，kube-state-metrics的重点是从Kubernetes的对象状态生成全新的指标。

最初，kube-state-metrics是作为Heapster的另一个指标数据源而开发的，Heapster只需要获取、格式化和转发原本就存在的指标，特别是Kubernetes组件的指标，并将它们写入接收器即可。相比之下，kube-state-metrics在内存中保存了Kubernetes系统状态的完整快照，并不断生

成基于它的新指标，但不负责在任何地方导出指标，那是Heapster的任务。

不过，有些监控系统（如Prometheus）根本不使用Heapster进行指标数据收集，而是自我实现了特有的收集方式，因此，将kube-state-metrics作为单独的项目，可以让Prometheus这类监视系统访问也能使用由它提供的指标。目前，kube-state-metrics主要负责为CronJob、Deployment、PersistentVolumeClaim等各类型的资源对象生成指标数据，各类资源类型生成的指标的功能及用法请参考相关的文档，地址为 <https://github.com/kubernetes/kube-state-metrics/tree/master/Documentation>。

将工作目录切换到克隆至本地的Kubernetes项目源码的cluster/addons/prometheus目录中，将所有文件名以kube-state-metrics相关的配置清单创建于集群中即能完成部署：

```
~]$ cd kubernetes/cluster/addons/prometheus/  
~]$ for file in kube-state-*; do kubectl apply -f $file; done
```

部署完成后会于kube-system名称空间创建一个名为kube-state-metrics的Deployment控制器对象，其所生成的Pod对象中的容器应用将监听于8080和8081端口以提供指标数据，除此之外，还会在kube-system名称空间中创建一个同名的Service对象为客户端提供固定的访问入口，用户也可以通过任何类型的HTTP客户端程序测试访问其原始格式的指标及数据。例如，启动一个Pod客户进行访问测试，命令如下：

```
~]$ kubectl run client-$RANDOM --image=cirros -it --rm -- sh  
/ # curl -s kube-state-metrics.kube-system:8080/metrics | tail  
# HELP kube_service_spec_type Type about service.  
# TYPE kube_service_spec_type gauge  
kube_service_spec_type{namespace="default",service="kubernetes",type="ClusterIP"}  
1  
.....
```

分别通过8080和8081端口进行确认能各自得到一系列的指标数据，即表示kube-state-metrics准备就绪。

2.Exporter及Node Exporter

Prometheus通过HTTP周期性抓取指标数据，监控目标上用于接收并响应数据抓取请求的组件统称为exporter。目前，Prometheus项目及社区提供了众多现成可用的exporter，分别用于监控操作系统、数据库、硬件设备、消息队列、存储系统等类型的多种目标对象。



提示 Prometheus的常用exporter列表位于<https://prometheus.io/docs/instrumenting/exporters/>。

在监控目标系统上，Exporter将收集的数据转化为文本格式，并通过HTTP对外暴露相应的接口，它不会局限于任何编程语言或实现方式，只需要它能够接收来自Prometheus服务器端的数据抓取请求，并以特定格式的数据进行响应，因此它尤其适用于容器环境。

- Exporter的响应内容以行为单位，其中，以“#HELP”开头的是注释信息用于提供指标说明，以“#TYPE”开头的注释行用于指明当前指标的类型，它可以是counter、gauge、histogram或summary其中之一；空白行将被忽略。

- 每一个非注释行都是一个键值类型的数据，其键即是指标，相应的值是当前指标本次响应的float类型的数据；若返回的值有两个，则最后一个会被当作时间戳；响应值必须要用双引号进行引用，否则即为格式错误。

Prometheus为监控类UNIX操作系统提供了一个专用的node_exporter程序，它能够收集多种系统级指标，如conntrack、cpu、diskstats、meminfo、filesystem、netstat和socketstats等数十种。node_exporter运行为守护进程监听于节点上的9100端口，通过URL路径/metrics提供指标数据，Prometheus服务器可在配置文件中静态配置监控每一个运行node_exporter的目标主机，也能够使用基于文件、Consul、DNS、Kubernetes等多种服务发现机制动态添加监控对象。

事实上，监控Kubernetes集群时，每个节点本身就能通过kubelet或cAdvisor提供符合Prometheus规范的节点指标数据，因此也可以不安装node_exporter程序，甚至Kubernetes为部署Prometheus服务器提供的配置清单中也没有加载各节点exporter上的指标。不过，用户也完全可以部署并自行配置通过node_exporter进程节点进行监控。

在此前克隆的Kubernetes源码目录的cluster/addons/prometheus路径下执行如下命令便可完成在集群中各节点上部署并运行node_exporter，它将以DaemonSet控制器对象于各节点中运行一个相应的Pod资源：

```
~]$ cd kubernetes/cluster/addons/prometheus/
~]$ for file in node-exporter-*; do kubectl apply -f $file; done
```

node_exporter相关的各Pod对象共享使用其所在节点的网络名称空间，它直接监听于相关地址的9100端点，因此可直接对其中某一个节点发起请求测试：

```
~]$ curl node01.ilinux.io:9100/metrics
.....
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.38784768e+08
```

上面显示的命令响应结果中，指标名称为process_virtual_memory_bytes，相应值为float格式，前面两行分别是帮助信息及指标类型说明。

3.告警系统Alertmanager

Prometheus的告警功能由两个步骤实现，首先是Prometheus服务器根据告警规则将告警信息发送给Alertmanager，而后由Alertmanager对收到的告警信息进行处理，包括去重、分组并路由到告警接收端，支持的形式包括slack、email、pagerduty、hitchat和WebHook等接口上的联系人信息。

下面的配置文件片段取自kubernetes源码目录中的cluster/addons/prometheus/alertmanager-configmap.yaml文件，它是一个ConfigMap对象，用于为运行于Pod中的Alertmanager提供配置信息：

```
global: null
receivers:
- name: default-receiver
route:
  group_interval: 5m
  group_wait: 10s
  receiver: default-receiver
  repeat_interval: 3h
```

上面的配置信息非常简洁，全局配置参数（`global`）为空，告警接收端（`receivers`）只有一个`default-receiver`，路由（`route`）逻辑是将告警信息暂存5分钟后进行发送，且每隔3小时重复一次。实际应用中需要提供的配置请读者参考Alertmanager的相关文档。

Prometheus附件的配置清单中，Alertmanager的Deployment对象配置容器使用了持久存储卷，它定义在Pod模板中，存储卷类型为`persistentVolumeClaim`，因此部署前需要确保能为其提供使用的存储卷。

另外，Alertmanager的守护进程通过9093端口提供了一个Web UI，用于检查和过滤告警信息。如需于Kubernetes集群之外访问此接口，这里选择将其Service对象设定为NodePort类型，并绑定固定的30093端口，配置片断如下所示：

```
spec:
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 9093
      nodePort: 30093
  selector:
    k8s-app: alertmanager
  type: "NodePort"
```

待条件满足后，即可部署Alertmanager：

```
-]$ cd kubernetes/cluster/addons/prometheus/
-]$ for file in node-exporter-*; do kubectl apply -f $file; done
```

部署完成后打开其Web UI即可过滤和查看告警信息，主界面如图14-10所示。

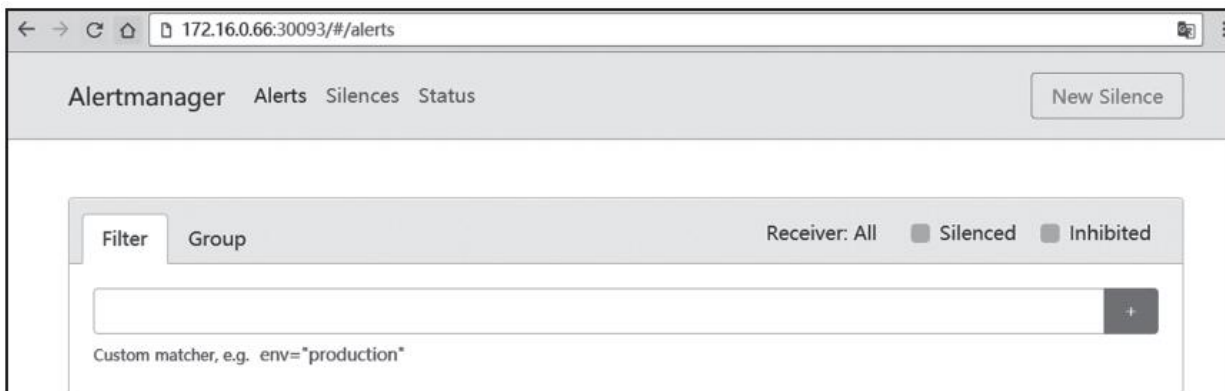


图14-10 Alertmanager的Web UI

4.部署Prometheus服务器

Prometheus服务器是整个监控系统的核心，它通过各exporter周期性采集指标数据，存储于本地的TSDB（Time Series Database）后端存储系统中，并通过PromQL向客户端提供查询接口。

由Kubernetes项目提供的附件配置清单中，Prometheus使用StatefulSet配置对象，各Pod对象通过volumeClaimTemplates对象申请使用持久存储卷，因此部署前需要确保能为其提供使用的存储卷。另外，Prometheus也提供了Web UI，通过9090端口输出，测试使用时，可以配置其通过NodePort类型的Service对象进行输出：

```
~]$ cd kubernetes/cluster/addons/prometheus/
~]$ for file in prometheus-*; do kubectl apply -f $file; done
```

确认各相关资源对象得以正确创建，且相关的Pod都能正常运行无误后即完成部署。此时，于Prometheus的Web GUI的Targets状态页面中可看到相关节点的发现结果，如图14-11所示。

The screenshot shows the Prometheus web interface at the URL 172.16.0.66:30090/targets. The 'Status' menu is open, showing options like Runtime & Build Information, Command-Line Flags, Configuration, Rules, Targets, and Service Discovery. The 'Targets' section is active, displaying a table of monitoring targets. The first section, 'kubernetes-apiservers (1/1)', shows one target with endpoint 'https://172.31.143.64:6443/metrics', state 'UP', and label 'instance="172.31.143.64:6443"'. The second section, 'kubernetes-nodes-cadvisor (4/4 up)', shows two targets. The first target has endpoint 'https://172.31.143.64:10250/metrics/cadvisor', state 'UP', and labels including 'beta_kubernetes_io_arch="amd64"', 'beta_kubernetes_io_os="linux"', 'instance="master.linux.io"', and 'kubernetes_io_hostname="master.linux.io"'. The second target has endpoint 'https://172.31.143.65:10250/metrics/cadvisor', state 'UP', and labels including 'beta_kubernetes_io_arch="amd64"', 'beta_kubernetes_io_os="linux"', 'instance="node01.linux.io"', and 'kubernetes_io_hostname="node01.linux.io"'. The 'Last Scrape' column shows the time since the last successful scrape, and the 'Error' column is empty for all targets.

Endpoint	State	Labels	Last Scrape	Error
https://172.31.143.64:6443/metrics	UP	instance="172.31.143.64:6443"	35.846s ago	
kubernetes-nodes-cadvisor (4/4 up) show less				
https://172.31.143.64:10250/metrics/cadvisor	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="master.linux.io" kubernetes_io_hostname="master.linux.io" node role_kubernetes_io_master=""	8.009s ago	
https://172.31.143.65:10250/metrics/cadvisor	UP	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="node01.linux.io" kubernetes_io_hostname="node01.linux.io"	13.063s ago	

图14-11 Prometheus的监控目标

PromQL（Prometheus Query Language）是Prometheus专有的数据查询语言（DSL），其提供了简洁且贴近自然语言的语法实现了时序数据的分析计算能力。PromQL表现力丰富，支持条件查询、操作符，并且内建了大量内置函数，可供客户端针对监控数据的各种维度进行查询。PromQL的使用方式请参考相关的文档或书籍。

14.3.3 自定义指标适配器k8s-prometheus-adapter

Prometheus提供的PromQL接口无法直接作为自定义指标数据源，它并非Kubernetes系统的聚合API服务器，它们之间还需要一个中间层“Kubernetes Custom Metrics Adapter for Prometheus”，目前最流行的自定义指标适配器是托管于Github之上的k8s-prometheus-adapter项目。

自定义的Kubernetes API Server必须要基于HTTPS与客户端进行通信，因此为了便于部署，这些程序大多数会生成一个自签证书。不过，k8s-prometheus-adapter项目提供的配置清单要使用一个名为cm-adapter-serving-certs的Secret对象加载由用户自行提供的证书。于是，部署之前需要以合理的方式生成证书并配置为Secret对象。这里选择以Master主机系统管理员的身份基于Kubernetes的CA为其签署一个自制证书：

```
~]# cd /etc/kubernetes/pki/
~]# (umask 077; openssl genrsa -out serving.key 2048)
~]# openssl req -new -key serving.key -out serving.csr -subj "/CN=serving"
~]# openssl x509 -req -in serving.csr -CA /etc/kubernetes/pki/ca.crt \
    -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out serving.crt -days 3650
```

k8s-prometheus-adapter默认会部署在custom-metrics名称空间，因此需要将相关证书和私钥作为此名称空间中的Secret对象。



注意 证书和私钥两个数据项的相应键名必须为serving.crt和serving.key。

```
~]$ kubectl create namespace custom-metrics
~]$ kubectl create secret generic cm-adapter-serving-certs -n custom-metrics \
    --from-file=serving.crt=./serving.crt --from-file=serving.key=./serving.key
```

确认创建完成后即可部署k8s-prometheus-adapter，部署清单位于项目源代码的deploy目录中。克隆到相关的代码并将资源创建于集群中即可：

```
~]$ git clone https://github.com/DirectXMan12/k8s-prometheus-adapter.git
~]$ kubectl apply -f k8s-prometheus-adapter/deploy/manifests/
```

资源创建过程中会在kube-aggregator上注册新的API群组custom.metrics.k8s.io，待相关的Pod对象转为正常运行状态之后即可通过kubectl api-versions命令确认其API接口注册的结果：

```
~]$ kubectl api-versions | grep custom
custom.metrics.k8s.io/v1beta1
```

向custom.metrics.k8s.io群组直接发送请求即可列出其可用的所有自定义指标，例如，这里使用“kubectl get--raw”命令进行测试，并使用jq命令过滤结果，仅列出指标名称：

```
~]$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1" | jq
'.resources[].name'
"pods/kube_pod_container_status_running"
"pods/kube_pod_info"
.....
```

另外，也可以直接通过API查看指定的Pod对象的相应指标及其值，例如，可以使用类似如下的命令列出kube-system名称空间中的所有Pod对象的文件系统占用率：

```
~]$ kubectl get --raw \
"/apis/custom.metrics.k8s.io/v1beta1/namespaces/kube-system/pods/*/memory_usage_bytes" | jq .
```

上面的命令会列出相应名称空间中所有Pod对象的内存资源占用状况，一个数据项的显示结果足以了解其响应格式，例如下面的内容截取自命令结果中关于coredns的Pod对象的相关输出：

```
{
  "describedObject": {
    "kind": "Pod",
    "namespace": "kube-system",
    "name": "coredns-78fcd6894-cft19",
    "apiVersion": "/__internal"
  },
  "metricName": "memory_usage_bytes",
  "timestamp": "2018-08-28T04:54:32Z",
```

```
"value": "40189952"  
}
```

14.4 自动弹性缩放

Deployment、ReplicaSet、Replication Controller或StatefulSet控制器资源管控的Pod副本数量支持手动方式的运行时调整，从而更好地匹配业务规模的实际需求。不过，手动调整的方式依赖于用户深度参与监控容器应用的资源压力并且需要计算出合理的值进行调整，存在一定程度的滞后性。为此，Kubernetes提供了多种自动弹性伸缩（Auto Scaling）工具，具体如下。

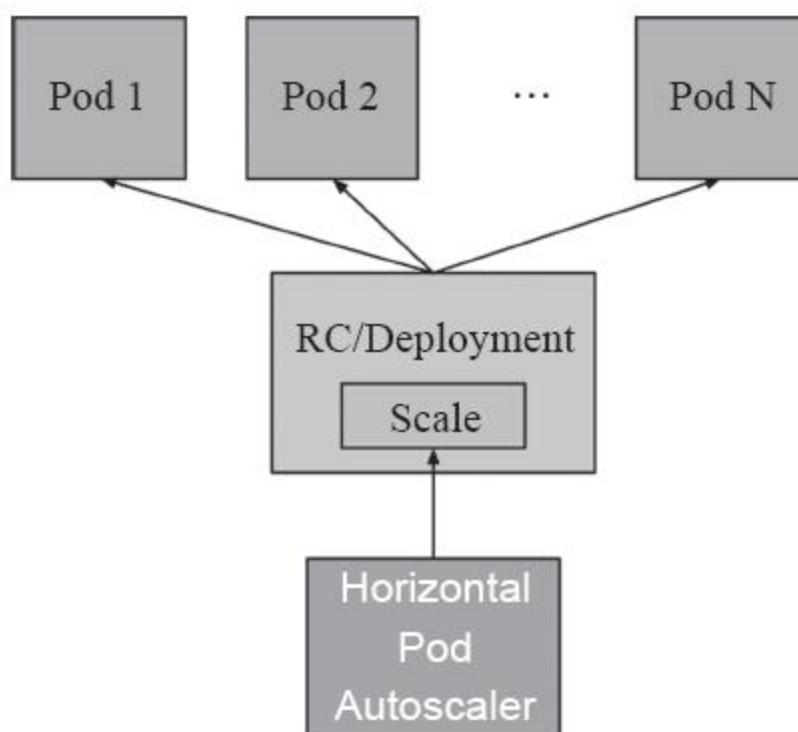


图14-12 HPA控制器示意图

·HPA：全称Horizontal Pod Autoscaler，一种支持控制器对象下Pod规模弹性伸缩的工具，目前有两个版本的实现，分别称为HPA和HPA（v2），前一种仅支持把CPU指标数据作为评估基准，而新版本支持可从资源指标API和自定义指标API中获取的指标数据。HPA控制器示意图如图14-12所示。

·CA：全称Cluster Autoscaler，是集群规模自动弹性伸缩工具，能自动增减GCP、AWS或Azure

·集群上部署的Kubernetes集群的节点数量，GA版本自Kubernetes 1.8起可用。

·VPA: 全称Vertical Pod Autoscaler，是Pod应用垂直伸缩工具，它通过调整Pod对象的CPU和内存资源需求量完成扩展或收缩，目前仍处于alpha阶段。

·AR: 全称Addon Resizer，是一个简化版本的Pod应用垂直伸缩工具，它基于集群中的节点数量来调整附加组件的资源需求量，当前仍处于beta级别。

14.4.1 HPA概述

尽管Cluster Autoscaler高度依赖基础云计算环境的底层功能，但HPA、VPA和AR可以独立于IaaS或PaaS云环境运行。HPA可作为Kubernetes API资源和控制器实现，它基于采集到的资源指标数据来调整控制器的行为，控制器会定期调整ReplicaSets或Deployment控制器对象中的副本数，以使得观察到的平均CPU利用率与用户指定的目标相匹配。

HPA自身是一个控制循环（control loop）的实现，其周期由controller-manager的--horizontal-pod-autoscaler-sync-period选项来定义，默认为30秒。在每个周期内，controller-manager将根据每个HPA定义中指定的指标查询相应的资源利用率。controller-manager从资源指标API（针对每个Pod资源指标）或自定义指标API（针对所有其他指标）中获取指标数据。

- 对于每个Pod资源指标（如CPU），控制器都将从HPA定位到的每个Pod的资源指标API中获取指标数据。若设置了目标利用率（target utilization）标准，则HPA控制器会计算其实际利用率

- （utilized/requests）。若设置的是目标原始值，则直接使用原始指标值。然后控制器获取所有目标Pod对象的利用率或原始值的均值（取决于指定的目标类型），并生成一个用于缩放所需副本数的比率。不过，对于未定义资源需求量的Pod对象，HPA控制器将无法定义该容器的CPU利用率，并且不会为该指标采取任何操作。

- 对于每个Pod对象的自定义指标，HPA控制器的功能与每个Pod资源指标的处理机制类似，只是它仅能够处理原始值而非利用率。

不过，使用HPA控制器管理Pod对象副本规模时，由于所评估指标的动态变动特性，副本数量可能会频繁波动，这种现象有时也称为“抖动”。从Kubernetes 1.6版本开始，集群管理员可以通过调整kube-controller-manager的选项值定义其变动延迟时长来缓解此问题。目前，默认的缩容延迟时长为5分钟，而扩容延迟时长为3分钟。

HPA控制器可以通过两种不同的方式获取指标：Heapster和REST客户端接口。使用直接Heapster获取指标数据时，HPA直接通过API服

务器的服务代理子资源向Heapster发起查询请求，因此，Heapster需要事先部署在群集上并在kube-system名称空间中运行。使用REST客户端接口获取指标时，需要事先部署好资源指标API及其API Server，必要时，还应该部署好自定义指标API及其相关的API Server。

HPA是Kubernetes autoscaling API群组中的API资源，当前的稳定版本仅支持CPU自动缩放，它位于autoscaling/v1群组中。而测试版本包含对内存和自定义指标的扩展支持，测试版本位于API群组autoscaling/v2beta1之中。

另外，自Kubernetes 1.6版本起还为HPA增加了基于多个指标的扩展支持，用户可以使用autoscaling/v2beta1 API版本为HPA配置多个指标以控制规模伸缩。运行时，HPA控制器将评估每个指标，并基于每个指标分别计算出各自控制下的新的Pod规模数量，结果值最大的即为采用的新规模标准。

14.4.2 HPA（v1）控制器

HPA也是标准的Kubernetes API资源，其基于资源配置清单的管理方式同其他资源相同。另外，它还有一个特别的“`kubectl autoscale`”命令用于快速创建HPA控制器。例如，首先创建一个名为myapp的Deployment控制器，而后通过一个同名的HPA控制器自动管控其Pod副本规模：

```
~]$ kubectl run myapp --image=ikubernetes/myapp:v1 --replicas=2 \
  --requests='cpu=50m,memory=256Mi' --limits='cpu=50m,memory=256Mi' \
  --labels='app=myapp' --expose --port=80
~]$ kubectl autoscale deploy myapp --min=2 --max=5 --cpu-percent=60
```

通过命令创建的HPA对象隶属于“`autoscaling/v1`”群组，因此，它仅支持基于CPU利用率的弹性伸缩机制，可从Heapster或Metrics Service获得相关的指标数据。下面的命令用于显示HPA控制器的当前状态：

```
~]$ kubectl get hpa myapp -o yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  .....
spec:
  maxReplicas: 5
  minReplicas: 2
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: myapp
  targetCPUUtilizationPercentage: 60
status:
  currentCPUUtilizationPercentage: 0
  currentReplicas: 2
  desiredReplicas: 2
```

HPA控制器会试图让Pod对象相应资源的占用率无限接近设定的目标值。例如，向myapp-svc的NodePort发起持续性的压力测试式访问请求，各Pod对象的CPU利用率将持续上升，直到超过目标利用率边界的60%，而后触发增加Pod对象副本数量。待其资源占用率下降到必须要降低Pod对象的数量以使得资源占用率靠近目标设定值时，即触发Pod副本的终止操作，如图14-13所示。



图14-13 HPA监控指标计算

下面是HPA控制器myapp在其详细信息“Events”中的一段内容，它显示了因资源占用率过高而增加Pod副本数量，以及资源占用率较低而降低Pod副本数量的部分操作事件：

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	SuccessfulRescale	39m (x3 over 1h)	horizontal-pod-autoscaler	New size: 5; reason: cpu resource utilization (percentage of request) above target
Normal	SuccessfulRescale	8m (x2 over 42m)	horizontal-pod-autoscaler	New size: 4; reason: All metrics below target
Normal	SuccessfulRescale	3m	horizontal-pod-autoscaler	New size: 2; reason: All metrics below target

当然，用户可以通过资源配置清单定义HPA（v1）控制器资源，其spec字段嵌套使用的属性字段主要包含maxReplicas、minReplicas、scaleTargetRef和targetCPUUtilization-Percentage几个，其使用方式请参考相应的文档获取。尽管CPU资源利用率可以作为规模伸缩的评估标准，但大多数时候，Pod对象所面临的访问压力未必会直接反映到CPU之上。

14.4.3 HPA（v2）控制器

HPA（v2）控制器支持基于核心指标CPU和内存资源以及基于任意自定义指标资源占用状态实现应用规模的自动弹性伸缩，它从metrics-server中请求查看核心指标，从k8s-prometheus-adapter一类的自定义API中获取自定义指标数据，如图14-14所示。

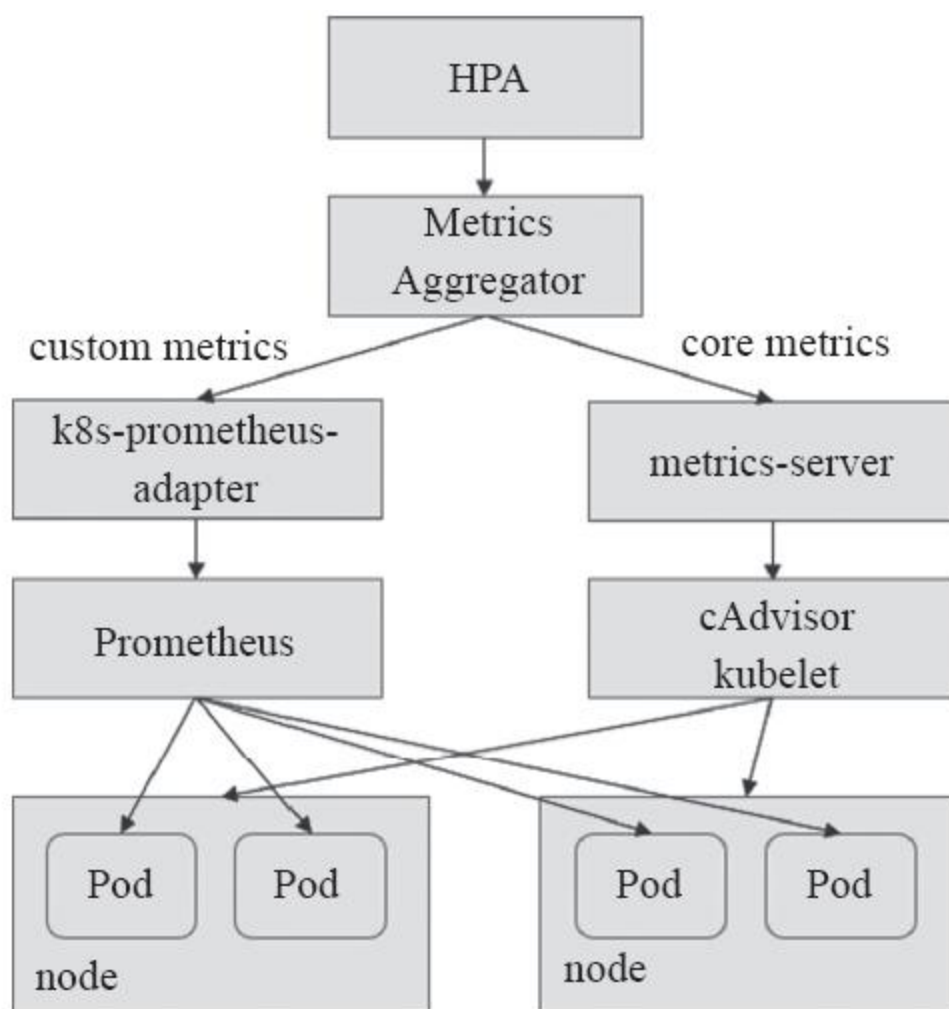


图14-14 HPA（v2）的数据指标获取方式

下面是一个HPA（v2）控制器的资源配置清单示例（hpa-v2-resources.yaml），它使用资源指标API获取两个指标CPU和内存资源的使用状况，并与其各自的设定目标进行比较，计算得出所需要的副本数量，两个指标计算的结果中数值较大的胜出：

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: myapp
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 50Mi
```

将配置清单中的资源创建于集群中即可进行应用规模伸缩测试，其测试方式与14.4.2节中的过程相似，这里不再给出其具体的过程。细心的读者或许已经注意到，HPA（v2）尚且处于beta阶段，将来它有可能会发生部分改变，但目前的特性足以支撑起其核心功能。HPA（v2）控制器的创建语法遵循标准的API资源格式，由apiVersion、kind、metadata和spec字段组成，其中spec字段主要嵌套使用如下几个字段。

·minReplicas<integer>：自动伸缩可缩减至的Pod副本数下限。

·maxReplicas<integer>：自动伸缩可扩展至的Pod副本数上限，其值不能低于min-Replicas属性值。

·scaleTargetRef<object>：要缩放的目标资源，以及应该收集指标数据并更改其副本数量的Pod对象，引用目标对象时，主要会用到三个嵌套属性—apiVersion、kind和name。

·metrics<[]object>：用于计算所需Pod副本数量的指标列表，每个指标单独计算其所需的副本数，将所有指标计算结果中的最大值作为最终采用的副本数量。计算时，以资源占用率为例，所有现有Pod对象的资源占用率之和除以目标占用率所得的结果即为目标Pod副本数，因此增加Pod副本数量必然地会降低各Pod对象的资源占用率。

`metrics`字段值是对象列表，它由要引用的各指标的数据源及其类型构成的对象组成。

- `external`: 用于引用非附属于任何对象的全局指标，甚至可以基于集群之外的组件的指标数据，如消息队列的长度等。

- `object`: 引用描述集群中某单一对象的特定指标，如Ingress对象上的`hits-per-second`等。定义时，嵌套使用`metricName`、`target`和`targetValue`分别用于指定引用的指标名称、目标对象及指标的目标值。

- `Pods`: 引用当前被弹性伸缩的Pod对象的特定指标，如`transactions-processed-per-second`等，各Pod对象的指标数据取平均值后与目标值进行比较。定义时，嵌套使用`metricName`和`targetAverageValue`分别指定引用的指标名称和目标平均值。

- `resource`: 引用资源指标，即当前被弹性伸缩的Pod对象中容器的`requests`和`limits`中定义的指标（CPU或内存资源）。定义时，嵌套使用`name`、`targetAverageUtilization`和`targetAverageValue`分别用于指定资源的名称、目标平均利用率和目标平均值。

- `type`: 表示指标源的类型，其值可为`Objects`、`Pods`或`Resource`。

基于自定义指标API中的指标配置HPA对象时，其指标的来源途径还包括`Pods`、`object`和`external`等，这其中又以Pod或引用的特定`object`的指标调用居多。下面通过一个示例来说明其用法，并测试其扩缩容的效果。

镜像文件`ikubernetes/metrics-app`在运行时会启动一个简单的Web服务器，它通过`/metrics`路径输出了`http_requests_total`和`http_requests_per_second`两个指标。下面是使用此镜像文件创建的Deployment控制器资源配置清单示例（`metrics-app.yaml`），它与此前正常使用的Deployment资源定义并无二致，除了特别附加的注解信息，包括其中的“`prometheus.io/scrape: "true"`”是使Pod对象能够被Prometheus采集相关指标的关键配置：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
```

```

    app: metrics-app
    name: metrics-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: metrics-app
  template:
    metadata:
      labels:
        app: metrics-app
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "80"
        prometheus.io/path: "/metrics"
    spec:
      containers:
      - image: ikubernetes/metrics-app
        name: metrics-app
        ports:
        - name: web
          containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: metrics-app
  labels:
    app: metrics-app
spec:
  ports:
  - name: web
    port: 80
    targetPort: 80
  selector:
    app: metrics-app

```

创建资源于集群中，启动一个专用的测试客户端Pod，在命令行中向创建出的Service端点的/metrics发起访问请求即可看到它输出的Prometheus兼容格式的指标及数据：

```

~]$ kubectl run client -it --image=cirros --rm -- /bin/sh
/ # curl metrics-app/metrics
# HELP http_requests_total The amount of requests in total
# TYPE http_requests_total counter
http_requests_total 660
# HELP http_requests_per_second The amount of requests per second the latest ten
seconds
# TYPE http_requests_per_second gauge
http_requests_per_second 0.5
/ #

```

命令结果中返回了所有的指标及其数据，每个指标还附带了通过注释行提供的帮助（**HELP**）信息和类型（**TYPE**）说明。Prometheus通过服务发现机制发现新创建的Pod对象，根据注解提供的配置信息或默认配置识别各个指标并纳入采集对象，而后由k8s-prometheus-adapter将这些指标注册到Kubernetes的自定义指标API中，提供给HPA（v2）控制器和Kubernetes调度器等作为调度评估参数使用。当然，用户也可以直接向自定义指标API接口发起请求。

下面的资源配置清单示例（metrics-app-hpa.yaml）用于自动弹性伸缩前面基于metrics-app.yaml创建的metrics-app相关的Pod对象副本数量，其伸缩标准是接入HTTP请求报文的速率，具体的数据则经由各现有相关Pod对象的http_requests指标的平均数据同目标速率800m（即0.8个/秒）的比较来进行判定：

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: metrics-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: metrics-app
  # autoscale between 2 and 10 replicas
  minReplicas: 2
  maxReplicas: 10
  metrics:
    # use a "Pods" metric, which takes the average of the
    # given metric across all pods controlled by the autoscaling target
    - type: Pods
      pods:
        # use the metric that you used above: pods/http_requests
        metricName: http_requests
        # target 500 milli-requests per second,
        # which is 1 request every two seconds
        targetAverageValue: 800m
```

创建资源于集群中，而后启动一个测试客户端发起持续性测试请求，模拟压力访问以便其指标数据能够满足扩展规模之需：

```
~]$ kubectl run client -it --image=cirros --rm -- /bin/sh
/ # while true; do curl http://metrics-app; let i++; sleep 0.$RANDOM; done
```

持续测试十几分钟，结束后再等上一段时间，待平均请求速率降下来之后即可通过**HPA**控制器的详细信息了解其规模变动状况，然后将会得到类似如下规模变动的相关信息：

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	SuccessfulRescale	18m	horizontal-pod-autoscaler	New size: 3;
reason:				
pods metric http_requests above target				
Normal	SuccessfulRescale	14m	horizontal-pod-autoscaler	New size: 5;
reason:				
pods metric http_requests above target				
Normal	SuccessfulRescale	11m	horizontal-pod-autoscaler	New size: 6;
reason:				
pods metric http_requests above target				
Normal	SuccessfulRescale	5m	horizontal-pod-autoscaler	New size: 5;
reason:				
All metrics below target				
Normal	SuccessfulRescale	2s	horizontal-pod-autoscaler	New size: 4;
reason:				
All metrics below target				

借助自定义指标自动弹性伸缩应用规模的机制，赋予了不同应用程序根据核心指标控制自身规模的能力，这是**Kubernetes**系统除敏捷部署（**Deployment**等控制器）功能之外又一极具特色的特性，其使得应用运维人员从繁重的日常监控和运维工作中解脱出来。

14.5 本章小结

本章介绍过第一代指标API及其实现方案Heapster之后，重点介绍了第二代监控架构体系、资源指标API、自定义指标API及其各自的解决方案，并对HPA及HPA（v2）的使用给予了详细说明。

- 指标API是HPA控制器、Dashboard和调度器依赖的基础组件，它们分别在指标数据的基础上实现了应用规模的弹性伸缩、指标数据的展示和Pod对象的调度。

- Heapster是第一代的指标API实现方案，也是Kubernetes系统曾经必不可少的核心附加组件之一。

- 新一代监控系统将指标划分为核心指标和自定义指标，并把API的定义同其实现分离开来。资源指标API的标准实现是Metrics Server，而自定义指标API的流行实现是Promethues及相应的适配器，如k8s-prometheus-adapter。

- HPA控制器第一代仅支持CPU指标数据，而第二代的HPA可基于各种核心指标和自定义指标实现应用规模的自动变动。

第15章 Helm程序包管理器

业务的容器化及微服务化过程基本上都是通过将单体大应用分解为多个小的服务并进行容器化编排运行来实现的，这种构建逻辑分解了单体应用的复杂性，让每个微服务都能够独立进行部署和扩展，实现了敏捷开发和运维。但另一方面，微服务化拆解巨大的单体应用为巨量的微服务程序，几乎必然地导致了应用管理复杂度的增加，例如，在Kubernetes系统之上，每个应用基本上都有着不止一个资源，而每个应用在不同的环境（如qa、test和prod等）中存在使用不同的配置参数的可能性等复杂问题。幸运的是，容器生态系统现在已经发展到了简便程度，因为有了Helm。

15.1 Helm基础

由前面章节中的应用部署过程可知，在Kubernetes系统上部署容器化应用时需要事先手动编写资源配置清单文件以定义资源对象，而且其每一次的配置定义基本上都是硬编码，基本上无法实现复用。对于较大规模的应用场景，应用程序的配置、分发、版本控制、查找、回滚甚至是查看都将是用户的噩梦。Helm可大大简化应用管理的难度。

简单来说，Helm就是Kubernetes的应用程序包管理器，类似于Linux系统之上的yum或apt-get等，可用于实现帮助用户查找、分享及使用Kubernetes应用程序，目前的版本由CNCF（Microsoft、Google、Bitnami和Helm社区）维护。它的核心打包功能组件称为chart，可以帮助用户创建、安装及升级复杂应用。

Helm将Kubernetes的资源（如Deployments、Services或ConfigMap等）打包到一个Charts中，制作并测试完成的各个Charts将保存到Charts仓库进行存储和分发。另外，Helm实现了可配置的发布，它支持应用配置的版本管理，简化了Kubernetes部署应用的版本控制、打包、发布、删除和更新等操作。Helm架构组件如图15-1所示。

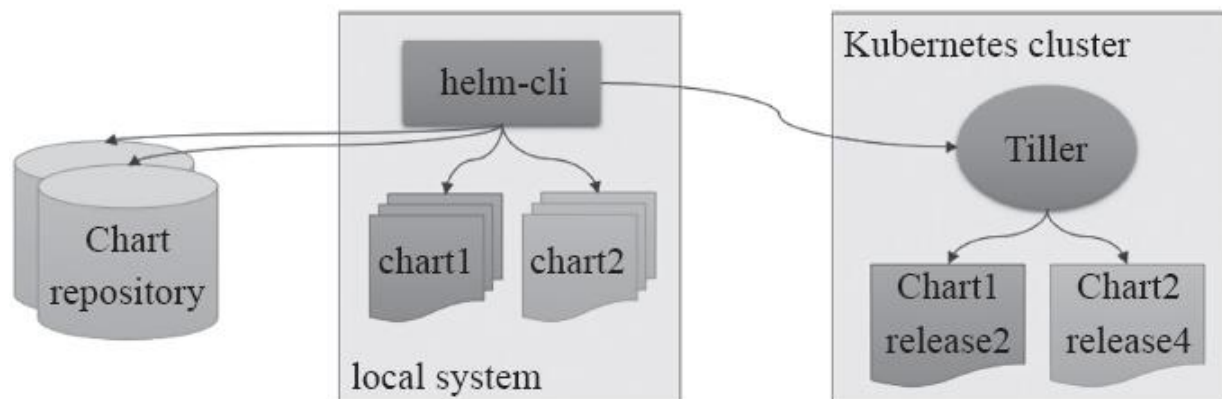


图15-1 Helm架构组件

简单来说，Helm其实就是一个基于Kubernetes的程序包（资源包）管理器，它将一个应用的相关资源组织成为Charts，并通过Charts管理程序包，其使用优势可简单总结为如下几个方面：

- 管理复杂应用： **Charts**能够描述哪怕是最复杂的程序结构，其提供了可重复使用的应用安装的定义。

- 易于升级： 使用就地升级和自定义钩子来解决更新的难题。

- 简单分享： **Charts**易于通过公共或私有服务完成版本化、分享及主机构建。

- 回滚： 可使用“**helm rollback**”命令轻松实现快速回滚。

15.1.1 Helm的核心术语

Helm将Kubernetes应用的相关配置组织为Charts，并通过它完成应用的常规管理操作。通常来说，使用Charts管理应用的流程包括从0开始创建Charts、将Charts及其相关的文件打包为归档格式、将Charts存储于仓库（repository）中并与之交互、在Kubernetes集群中安装或卸载Charts以及管理经Helm安装的应用的版本发行周期。因此，对Helm来说，它具有以下几个关键概念。

- Charts: 即一个Helm程序包，它包含了运行一个Kubernetes应用所需要的镜像、依赖关系和资源定义等，必要时还会包含Service的定义；它类似于APT的dpkg文件或者yum的rpm文件。

- Repository: Charts仓库，用于集中存储和分发Charts，类似于Perl的CPAN，或者Python的PyPI。

- Config: 应用程序实例化安装运行时使用的配置信息。

- Release: 应用程序实例化配置后运行于Kubernetes集群中的一个Charts实例；在同一个集群上，一个Charts可以使用不同的Config重复安装多次，每次安装都会创建一个新的Release。

事实上，Charts更像是存储于Kubernetes集群之外的程序，它的每次安装是指在集群中使用专用配置运行一个实例，执行过程有点类似于在操作系统上基于程序启动一个进程。

15.1.2 Helm架构

Helm主要由Helm客户端、Tiller服务器和Charts仓库（repository）组成，如图15-2所示。

Helm客户端是命令行客户端工具，采用Go语言编写，基于gRPC协议与Tiller server交互（见图15-2）。它主要完成如下任务。

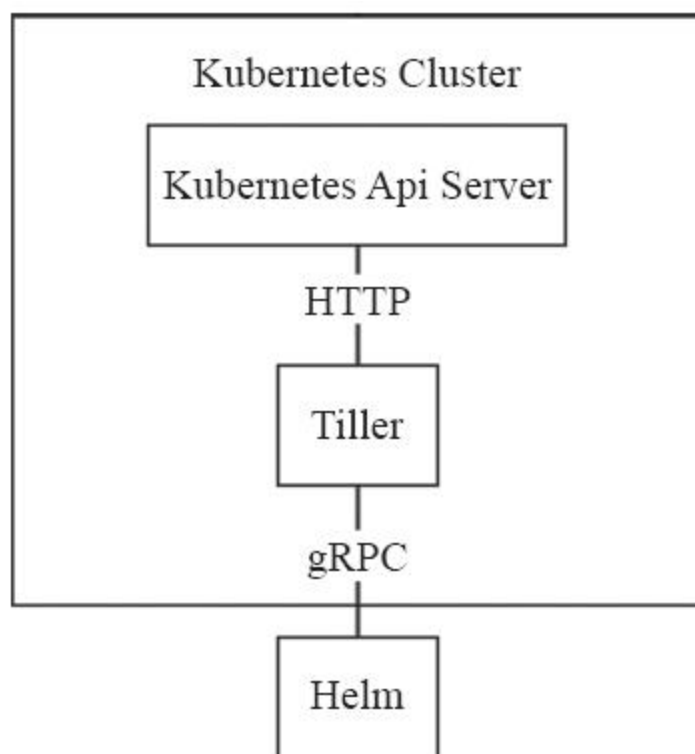


图15-2 Helm成员间通信

- 本地Charts开发。
- 管理Charts仓库。
- 与Tiller服务器交互：发送Charts以安装、查询Release的相关信息以及升级或卸载已有的Release。

Tiller server是托管运行于Kubernetes集群之中的容器化服务应用，它接收来自Helm客户端的请求，并在必要时与Kubernetes API Server进

行交互。它主要完成以下任务。

- 监听来自于Helm客户端的请求。
- 合并Charts和配置以构建一个Release。
- 向Kubernetes集群安装Charts并对相应的Release进行跟踪。
- 升级和卸载Charts。

通常，用户于Helm客户端本地遵循其格式编写Charts文件，而后即可部署于Kuber-netes集群之上运行为一个特定的Release。仅在有需要分发需求时，才应该将同一应用的Charts文件打包成归档压缩格式提交到特定的Charts仓库。仓库既可以运行为公共托管平台，也可以是用户自建的服务器，仅供特定的组织或个人使用。

15.1.3 安装Helm Client

Helm的安装方式有两种：预编译的二进制程序和源码编译安装。这里先介绍预编译的二进制程序的安装方式，源码编译安装的方式将在15.1.4节的Tiller编译安装中一并说明。

Helm的每个发行版都提供了主流操作系统的专用版本，主要包括Linux、Mac OS和Windows，用户安装前按需下载合用的平台上的相关发行版本即可。Helm项目托管在GitHub之上，项目地址为<https://github.com/kubernetes/helm>。

安装之前首先下载合用版本的压缩包并将其展开，本示例中使用的是v2.9.1的版本。执行具体命令时，需要替换为下载到的版本：

```
~]$tar -zxvf helm-v2.9.1-linux-amd64.tgz
```

而后，将其二进制程序文件复制或移动到系统PATH环境变量指向的目录中即可，如/usr/local/bin/目录（管理员用户才有写入文件至此目录的权限）：

```
~]$sudo mv linux-amd64/helm/usr/local/bin/
```

Helm的各种管理功能均可通过其子命令完成，获取其使用帮助，直接使用“help”子命令即可：

```
~]$helm help
```

需要注意的是，Helm的运行依赖于本地安装并配置完成的kubectl方能与运行于Kubernetes集群之上的Tiller服务器进行通信，因此，运行Helm的节点也应该是可以正常使用kubectl命令的主机，或者至少是有着可用kubeconfig配置文件的主机。Mac OS或Windows系统上的安装方式请参考Helm官方文档。

15.1.4 安装Tiller server

Tiller是Helm的服务器端，一般应该运行于Kubernetes集群之上，不过，出于研发使用的目的，也可以将其部署于本地，且需要能够与远程Kubernetes集群正常通信，这里选择将其托管运行于集群之上。另外，对于了RBAC授权插件的Kubernetes集群来说，还需要事先创建相关的ServiceAccount才能进行安装。

下面的资源配置清单示例中（tiller-rbac.yaml）定义了一个名为tiller的ServiceAccount，并通过ClusterRoleBinding将其绑定至集群管理员角色cluster-admin，从而使得它拥有集群级别所有的最高权限：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

将上面清单的内容保存于文件中，如tiller-rbac.yaml，而后执行命令以完成绑定：

```
-j$ kubectl apply -f tiller-rbac.yaml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

而后使用如下命令进行Tiller server环境的初始化，完成Tiller server安装：

```
~]$ helm init --service-account tiller
```



注意 `helm init`命令进行初始化时，Kubernetes集群会到 `gcr.io/kubernetes-helm/` 上获取需要的镜像，镜像标签同Helm的版本号。请确保Kubernetes集群能够正确访问此镜像仓库。

若有必要，可以基于SSL/TLS实现Helm和Tiller之间的通信，这就需要在部署Tiller时为其指定专用的选项`--tiller-tls-verify`，并为后续的所有Helm命令额外附加`--tls`选项。部署完成后可使用“`kubectl get pods`”命令确认Tiller Pod运行正常，如下面的命令所示：

```
~]$ kubectl get pods -n kube-system -l app=helm
```

NAME	READY	STATUS	RESTARTS	AGE
tiller-deploy-5c688d5f9b-gt65w	1/1	Running	0	4m

安装完成后，运行“`helm version`”命令即可显示客户端和服务端的版本号，若两者均显示正常，则表示安装成功。到此为止，运行于Kubernetes集群中的Tiller Server已经配置完成，管理员可按需实现后续的其他管理工作，例如，搜索可用的Charts，安装Charts到集群中等。

如果希望在安装时自定义一些参数以设定其运行机制，例如Tiller的版本或者在Kubernetes集群上的目标名称空间，则可以以类似如下方式使用命令：

- `--canary-image`：安装canary分支，即项目Master的分支。
- `--tiller-image`：安装指定版本的镜像，默认同Helm版本。
- `--kube-context`：安装至指定的Kubernetes集群。
- `--tiller-namespace`：安装至指定的名称空间，默认为kube-system。

此外，Tiller将数据存储于ConfigMap资源中，因此卸载后重新安装并不会导致数据丢失，必要时，管理员尽可放心重新安装或升级。卸载Tiller的方法常用的有两种方式。

1) `kubectl delete deployment tiller-deploy--namespace kube-system`

2) `helm reset`

至此，Helm和Tiller的安装设定工作已经完成，接下来便可尝试使用Helm带来的诸多便捷之处。

15.1.5 Helm快速入门

Charts是Helm的程序包，它们存储于Charts仓库中。Kubernetes官方的Charts仓库保存了一系列精心制作和维护的Charts，仓库的默认名称为“stable”。安装Charts到Kubernetes集群时，Helm首先会到Kubernetes官方的Charts仓库中获取到相关的Charts，而后将其安装并创建为Release。



提示 Helm的官方仓库为<https://kubernetes-charts.storage.googleapis.com>，进行后续的操作之前请确保拥有访问此站点的能力。

“helm repo”相关的命令可用于管理使用的Charts仓库，其update子命令能够更新使用的默认仓库的元数据信息，其命令及执行结果如下所示：

```
~]$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. * Happy Helming! *
```

“helm search”命令可列出stable仓库中维护的所有Charts的列表，如下面命令结果中列出的部分Charts：

```
~]$ helm search
NAME                CHART VERSION  APP VERSION  DESCRIPTION
stable/coredns      0.9.0          1.0.6        CoreDNS is a DNS server that
chains plugins and...
stable/docker-registry 1.4.0          2.6.2        A Helm chart for Docker Registry
stable/etcd-operator 0.7.7          0.7.0        CoreOS etcd-operator Helm chart
for Kubernetes
stable/gitlab-ce      0.2.1          GitLab       Community Edition
stable/grafana        1.9.0          5.1.2        The leading tool for querying
and visualizing t...
stable/jenkins        0.16.1         2.107        Open source continuous inte-
gration server. It s...
stable/kubernetes-dashboard 0.6.8          1.8.3        General-purpose web UI for Kub-
ernetes clusters
stable/mysql          0.5.0          5.7.14       Fast, reliable, scalable, and
easy to useopen-...
.....
```

也可以为`search`命令添加一个过滤器，仅列出符合条件的Charts，例如下面的命令以`redis`为过滤条件，仅显示与`redis`相关的所有Charts：

```
~]$ helm search redis
NAME                                CHART VERSION  APP VERSION  DESCRIPTION
stable/prometheus-redis-exporter 0.1.1          0.16.0       Prometheus exporter for Redis
metrics
stable/redis                        3.3.0          4.0.9       Open source, advanced key-
value store. It is of...
```

“`helm inspect`”命令能够打印出指定的Charts的详细信息：

```
~]$ helm inspect stable/redis
appVersion: 4.0.9
description: Open source, advanced key-value store. It is often referred to as a
data
structure server since keys can contain strings, hashes, lists, sets and sorted
sets.
.....
```

安装指定的Charts为Kubernetes集群的Release，可使用“`helm install`”命令进行，例如若需要安装`stable/redis`，则为命令行使用“-n”选项指定Release名称即可。当然，也可以先执行安装测试，例如：

```
~]$ helm install stable/redis -n redis --dry-run
NAME:  redis
```

若无错误信息返回，则移除`--dry-run`选项即可进入安装流程，命令会返回安装过程的执行步骤，以及最后的注意信息，它们通常是应用的使用帮助，因此是需要特别注意的内容：

```
~]$ helm install stable/redis -n redis
NAME:  redis
LAST DEPLOYED: Thu May 17 13:10:11 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME    TYPE    DATA  AGE
redis  Opaque  1      0s

==> v1/Service
```

```
NAME          TYPE          CLUSTER-IP      EXTERNAL-IP  PORT(S)    AGE
redis-master  ClusterIP     10.105.63.22    <none>       6379/TCP   0s
redis-slave   ClusterIP     10.105.114.60   <none>       6379/TCP   0s
.....
NOTES:
** Please be patient while the chart is being deployed **
Redis can be accessed via port 6379 on the following DNS names from within your
cluster:

redis-master.default.svc.cluster.local for read/write operations
redis-slave.default.svc.cluster.local for read-only operations
.....
```

多数应用的Charts都可以接受用户提供的配置参数进行特定的场景化部署，所有可配置参数都会通过Charts模板提供，用户把需要提供参数值的配置选项存储在YAML格式的配置文件并通过-f命令行选项加载即可。

若要列出已经安装生成的Release，则需要使用“helm list”命令：

```
~]$ helm list
NAME      REVISION  UPDATED              STATUS      CHART          NAMESPACE
redis     1         Thu May 17 13:10:11 2018  DEPLOYED   redis-3.3.0    default
```

而要删除Release，则使用“helm delete”命令即可。

```
~]$ helm delete redis
release "redis" deleted
```

升级或回滚应用，要分别使用“helm upgrade”和“helm rollback”命令，而且还可以使用“helm history”命令获取指定的Release变更的历史。不过，若默认仓库中不存在所需要的某Charts，而对用户来说该Charts是日常部署任务，则用户可以自行编写Charts并分享。事实上，helm install命令支持基于多种安装源进行应用部署，这包括Charts仓库、本地的Charts压缩包、本地Charts目录，甚至是指定某个Charts的URL。

15.2 Helm Charts

Charts是Helm使用的Kubernetes程序包打包格式，一个Charts就是一个描述一组Kubernetes资源的文件的集合。事实上，一个单独的Charts既能用于部署简单应用，例如一个memcached Pod，也能部署复杂的应用，如由HTTP服务器、DB服务器、Cache服务器和应用程序服务器等共同组成的Web应用栈。

从物理的角度来描述，Charts是一个遵循特定规范的目录结构，它能够打包成为一个可用于部署的版本化归档文件。

15.2.1 Charts文件组织结构

一个Charts就是按特定格式组织的目录结构，目录名即为Charts名，目录名称本身不包含版本信息。例如，一个jenkins Charts的目录结构应该如下所示：

```
jenkins/  
  Chart.yaml  
  LICENSE  
  README.md  
  requirements.yaml  
  values.yaml  
  charts/  
  templates/  
  templates/NOTES.txt
```

目录结构中除了charts/和templates/是目录之外，其他的都是文件。它们的基本功用如下。

- Chart.yaml: 当前Charts的描述信息，yaml格式的文件。
- LICENSE: 当前Charts的许可证信息，纯文本文件；此为可选文件。
- README.md: 易读格式的README文件；可选。
- requirements.yaml: 当前Charts的依赖关系描述文件；可选。
- values.yaml: 当前Charts用到的默认配置值。
- charts/: 目录，存放当前Charts依赖到的所有Charts文件。
- templates/: 目录，存放当前Charts用到的模板文件，可应用于Charts生成有效的Kuber-netes清单文件。
- templates/NOTES.txt: 纯文本文件，Templates简单使用注解。

尽管Charts和Templates目录均为可选，但至少应该存在一个Charts依赖文件或一个模板文件。另外，Helm保留使用charts/和templates/目

录以及上面列出的文件名称，其他文件都将被忽略。

15.2.2 Chart.yaml文件组织格式

Chart.yaml用于提供Charts相关的各种元数据，如名称、版本、关键词、维护者信息、使用的模板引擎等，它是一个Charts必备的核心文件，主要包含以下字段。

- name: 当前Charts的名称，必选字段。
- version: 遵循语义化版本规范第2版的版本号，必选字段。
- description: 当前项目的单语句描述信息，可选字段。
- keywords: 当前项目的关键词列表，可选字段。
- home: 当前项目的主页URL，可选字段。
- sources: 当前项目用到的源码的来源URL列表，可选字段。
- maintainers: 项目维护者信息，主要嵌套name、email和URL几个属性组成；可选字段。
- engine: 模板引擎的名称，默认为gotpl，即go模板。
- icon: URL，指向当前项目的图标，SVG或PNG格式的图片；可选字段。
- appVersion: 本项目用到的应用程序的版本号，可选字段，且不必为语义化版本。
- deprecated: 当前Charts是否已废弃，可选字段，布尔型值。
- tillerVersion: 当前Charts依赖的Tiller版本号，可以是语义化版本号的范围，如“>2.4.0”；可选字段。

例如，下面的示例信息是rediss Charts中使用的Chart.yaml的内容，用户自行定义Charts编写相关文件时，要采用类似的文件格式：

appVersion: 4.0.9
description: Open source, advanced key-value store. It is often referred
to as a data structure server since keys can contain strings, hashes,
lists, sets and sorted sets.
engine: gotpl
home: <http://redis.io/>
icon: <https://bitnami.com/assets/stacks/redis/img/redis-stack-220x234.png>
keywords:
- redis
- keyvalue
- database
maintainers:
- email: containers@bitnami.com
name: bitnami-bot
name: redis
sources:
- <https://github.com/bitnami/bitnami-docker-redis>
version: 3.3.0

15.2.3 Charts中的依赖关系

Helm中的一个Charts可能会依赖不止一个其他的Charts，这种依赖关系可经requirements.yaml进行动态链接，也可直接存储于charts/目录中进行手动管理。不过，尽管手动管理依赖关系对个别管理场景也有着些许优势，但使用动态管理的方式却是推荐的首选方式。

1.requirements.yaml文件

requirements.yaml文件本质上只是一个简单的依赖关系列表，它具有类似如下定义中的格式中的可用字段：

```
dependencies:
- name:
  version:
  repository:
  alias:
  tags:
  condition:
  import-values:
    - child:
      parent:
```

上述示例中的字段基本可以见名知义，具体如下。

·name: 被依赖的Charts的名称。

·version: 被依赖的Charts的版本。

·repository: 被依赖的Charts所属的仓库及其URL；如果是非官方的仓库，则需要先用helm repo add命令将其添加进本地可用仓库。

·alias: 为被依赖的Charts创建一个别名，从而让当前Charts可以将所依赖的Charts对应到新名称，即别名；可选字段。

·tags: 默认情况下所有的Charts都会被装载，若给定了tags，则仅装载那些匹配到的Charts。

·**condition**: 类似于**tags**字段，但需要通过自定义的条件来指明要装载的charts。

·**import-values**: 导入子Charts中的值；被导入的值需要在子charts中导出。

如下所示的示例，是Wordpress Charts中定义的动态依赖关系：

```
dependencies:
- name: mariadb
  version: 2.1.1
  repository: https://kubernetes-charts.storage.googleapis.com/
  condition: mariadb.enabled
  tags:
    - wordpress-database
```

一旦依赖关系文件配置完成，即可使用“**helm dependency update**”命令更新依赖关系，并自动下载被依赖的Charts至charts/目录中。

2.Charts目录

若需要对依赖关系进行更多的控制，则所有被依赖到的Charts都能以手工方式直接复制到Charts目录中。一个被依赖到的Charts既可以是归档格式，也可以是展开的目录格式，不过，其名称不能以下划线（**_**）或点号（**.**）开头，此类文件会被Charts装载机自动忽略。

例如，Wordpress Charts依赖关系在其Charts目录中的反映类似如下所示：

```
charts/
├── mariadb
│   ├── Chart.yaml
│   ├── README.md
│   ├── templates
│   │   ├── configmap.yaml
│   │   ├── deployment.yaml
│   │   ├── _helpers.tpl
│   │   ├── NOTES.txt
│   │   ├── pvc.yaml
│   │   ├── secrets.yaml
│   │   ├── svc.yaml
│   │   ├── test-runner.yaml
│   │   └── tests.yaml
│   └── values.yaml
```


15.2.4 模板和值

Helm Charts模板（template）遵循Go模板语言格式，并支持50种以上的来自Spring库的模板函数附件，以及为数不少的其他专用函数。所有的模板文件都存储于Templates目录中，在当前Charts被Helm引用时，此目录中的所有模板文件都会传递给模板引擎进行处理。

模板文件中用到的值（value）有如下两种提供方式。

- 通过Charts的values.yaml文件提供，通常用于提供默认值。

- 在运行“helm install”命令时传递包含所需要的自定义值的YAML文件；此处传递的值会覆盖默认值。

下面的示例是Wordpress Charts中Deployment模板文件的部分内容：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: {{ template "fullname" . }}
  labels:
    app: {{ template "fullname" . }}
    chart: "{{ .Chart.Name }}" - "{{ .Chart.Version }}"
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: {{ template "fullname" . }}
    spec:
      containers:
        - name: {{ template "fullname" . }}
          image: "{{ .Values.image }}"
          imagePullPolicy: {{ default "" .Values.imagePullPolicy | quote }}
```

示例中模板相关的代码部分会由模板引擎运行，并生成相应的结果。供依赖的值可由values.yaml文件或由用户在运行helm install命令时通过选项提供，除此之外，char模板还包含一些固定的预定义值，如Release.Name、Release.Time、Release.Time、Release.Service、

Release.IsUpgrade、Release.IsInstall、Release.Revision、Chart.Name、Chart.Version、Files和Capabilities等。

而在values.yaml一类的文件中定义值（value）时，既可以将它们定义为全局作用域，也可以定义为仅供Charts目录下的某个Charts所使用。一般来说，上级Charts可访问下级Charts中的值，但下级Charts不能访问其上级Charts的值。

如下面示例中的内容，其中title属于全局作用域，max_connections和password则仅属于mysql Charts，port仅属于apache Charts：

```
title: "My WordPress Site" # Sent to the WordPress template

mysql:
  max_connections: 100 # Sent to MySQL
  password: "secret"

apache:
  port: 8080 # Passed to Apache
```

Go模板语法请参考godoc站点中的内容，地址为<https://godoc.org/text/template>。

15.2.5 其他需要说明的话题

定义Charts时还需要用到许可证文件（License）、自述文件（README.md）以及说明文件（NOTE.txt），其中说明文件需要为用户提供重要的使用帮助及注意事项等。基于Charts的格式规范，用户即可自定义相关应用程序的Charts，并将其通过仓库完成分享。

Charts中也支持使用文件来描述安装、配置、使用和许可证信息。一般说来，README文件必须为Markdown格式，因此其后缀名通常是“.md”，它一般应该包含如下内容。

- 当前Charts提供的应用或服务的描述信息。
- 运行当前Charts需要满足的条件。
- values.yaml文件中选项及默认值的描述。
- 其他任何有助于安装或配置当前Charts的有用信息。

另外，templates/NOTES.txt文件中的内容将会在Charts安装完成后予以输出，通常用于向用户提供当前Charts相关的使用或初始访问方式的信息。另外，使用“helm status”命令查看某Release的相关状态信息时，此文件中的内容也会输出。

15.2.6 自定义Charts

一个典型的服务类容器化应用通常会由Pod控制器（常用的为Deployment）、Service、ConfigMap、Secret、Ingress和PersistentVolumeClaim等资源对象组成，其中前两者基本上是必备的资源，后面三个则可按需进行定义。Helm Charts包含了这些组件的yaml格式的资源配置文件模板，它们能够通过values.yaml配置文件获取模板中的各个变量所需的“值”，甚至支持用户在部署操作的运行时进行配置。

1. 生成一个空Charts

创建一个新Charts的有效方式是使用“helm create”命令，它能够在新目录中创建一个名为mychart的新图表。例如，下面的命令会于命令执行的当前目录中创建一个名为mychart的子目录作为Charts存储路径：

```
~]$ helm create mychart
Creating mychart
```

此命令会初始化出一个空的Charts目录结构，它有着所需要的各个核心文件：

```
~]$ tree mychart/
mychart/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

由命令生成的各文件还有着各自应该具有的通用组织结构框架，例如，Chart.yaml文件的默认内容如下：

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for Kubernetes
name: mychart
version: 0.1.0
```

事实上，它甚至直接在`values.yaml`将要使用的镜像文件定义中为`nginx`生成了一个可直接安装容器化`Nginx`应用的`Charts`，其中的部分内容如下所示：

```
replicaCount: 1

image:
  repository: nginx
  tag: stable
  pullPolicy: IfNotPresent

service:
  type: ClusterIP
  port: 80
```

因此，用户仅需要在各文件现有框架的基础上按需进行修改即可定义出所需的`Charts`来。

2.修改Charts以部署自定义服务

这里以此前使用的容器应用“`ikubernetes/myapp: v1`”为示例来说明如何定义一个`Charts`。使用“`helm create`”命令生成的`Charts`会创建一个用于运行、由默认值提供的镜像的`Deployment`对象，这就意味着若要部署不同的应用仅通过修改`values.yaml`中的引用镜像文件即可。当然，必要时，用户还需要额外确认是否需要默认的`Ingress`对象的定义，以及需要额外用到存储资源。

例如，这里将`values.yaml`文件的内容修改为如下所示：

```
repository: ikubernetes/myapp
tag: v1
pullPolicy: IfNotPresent

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
```

```
annotations: {}
  # kubernetes.io/ingress.class: nginx
  # kubernetes.io/tls-acme: "true"
path: /
hosts:
  - chart-example.local
tls: []
# - secretName: chart-example-tls
#   hosts:
#     - chart-example.local

resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 500m
    memory: 512Mi

nodeSelector: {}

tolerations: []

affinity: {}
```

而后通过“**helm lint**”命令确认修改后的Charts是否遵循最佳实践且模板格式良好:

```
~]$ helm lint mychart
==> Linting mychart
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, no failures
```

多数情况下, “**helm lint**”命令报告的错误信息, 根据其错误提示中的行号信息即能定位出错误所在。确保一切问题都得以解决之后, 即可通过“**helm install**”命令调试运行以查看由Charts定义的容器化应用是否能够正确部署:

```
~]$ helm install --name myapp --dry-run --debug ./mychart --set service.type=NodePort
[debug] Created tunnel using local port: '40255'

[debug] SERVER: "127.0.0.1:40255"

[debug] Original chart version: ""
[debug] CHART PATH: /home/ik8s/charts/mychart

NAME:      myapp
REVISION:  1
RELEASED:  Tue Jun  5 16:14:45 2018
```

```
CHART: mychart-0.1.0
USER-SUPPLIED VALUES:
service:
  type: NodePort
.....
```

确认上述命令输出信息无误后，移除命令中的“--dry-run”选项后再次运行命令即可完成应用的部署：

```
~]$ helm install --name myapp ./mychart --set service.type=NodePort
NAME: myapp
LAST DEPLOYED: Tue Jun 5 16:20:22 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)          AGE
myapp-mychart       NodePort    10.108.65.88  <none>       80:30315/TCP     1s

==> v1/Deployment
NAME                DESIRED     CURRENT       UP-TO-DATE   AVAILABLE   AGE
myapp-mychart       1           1             1            0           0s

==> v1/Pod(related)
NAME                                   READY   STATUS              RESTARTS   AGE
myapp-mychart-67b9997c8b-nbsfk        0/1     ContainerCreating   0           0s

NOTES:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services myapp-mychart)
  export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT
```

而后，通过上述NOTES中的命令提示运行相关的命令获取访问端点后即可通过浏览器访问相应的服务：

```
~]$ export NODE_PORT=$(kubectl get --namespace default -o \
  jsonpath="{.spec.ports[0].nodePort}" services myapp-mychart)
~]$ export NODE_IP=$(kubectl get nodes --namespace default -o \
  jsonpath="{.items[0].status.addresses[0].address}")
~]$ echo http://$NODE_IP:$NODE_PORT
http://172.16.0.70:30376
```

而后通过浏览器访问测试所部署的myapp应用。

3.Charts仓库 (Repository)

至此，一个自定义的Charts即于本地设定完成，不过，它仅能用于本地访问。当然，用户也可以通过“helm package”命令将其打包为tar格式后分享给团队或者社区：

```
~]$ helm package ./mychart
Successfully packaged chart and saved it to: /root/charts/mychart-0.1.0.tgz
```

Helm将在工作目录中创建一个mychart-0.1.0.tgz包，它使用Chart.yaml文件中定义的元数据的名称和版本。通过将程序包作为参数传递给helm install，用户可以基于该程序包而不是本地目录进行安装：

```
~]$ helm install --name myapp2 mychart-0.1.0.tgz --set service.type=NodePort
```

为了使软件包更容易分享，Helm内置了从HTTP服务器安装软件包的支持。运行时，Helm读取服务器上托管的仓库索引，该索引描述了有哪些Charts程序包可用以及它们位于何处。使用“helm serve”命令即可运行本地仓库来输出本地创建的Charts：

```
~]$ helm serve
Regenerating index. This may take a moment.
Now serving you on 127.0.0.1:8879
```

此命令会占据当前终端，于是，另启一个终端即可测试访问本地仓库服务中的Charts：

```
~]$ helm search local
```

NAME	CHART VERSION	APP VERSION	DESCRIPTION
local/mychart	0.1.0	1.0	A Helm chart for Kubernetes

尽管Helm能够管理本地仓库，但创建好的Charts如需向外分享，就需要为其创建用于共享的仓库，并将计划共享的所有Charts都放置于仓库中。事实上，Charts仓库服务器就是HTTP服务器，任何能支持YAML文件及tar文件传输的HTTP服务器程序都可以作为仓库服务器使用。不过，一般来说，一个仓库应该有一个index.html主页用于描述当前仓库中的所有Charts及其元数据信息。然而，到目前为止，Helm并

不支持将Charts上传至仓库中，因为这样做会对仓库服务器程序多出很多额外的要求，并且也会增加其配置的复杂度。这就意味着，上传Charts到仓库中需要借助于其他手段来进行，如FTP或SSH协议等，另外，如需自建Charts仓库，则各流行的Web服务器程序基本上都能满足要求，如Apache或Nginx等。安全起见，通过互联网提供服务时，建议使用HTTPS的服务器提供仓库服务。

而在Helm客户端，仓库的管理需要使用“helm repo”命令进行，它可以添加、删除、列出、及索引仓库。例如，使用如下命令添加incubator仓库以便使用更多的Charts：

```
~]$ helm repo add incubator http://storage.googleapis.com/kubernetes-charts-incubator
"incubator" has been added to your repositories
```

列出所有可用的仓库列表，可以使用“helm repo list”命令进行：

```
~]$ helm repo list
NAME      URL
stable    https://kubernetes-charts.storage.googleapis.com
local     http://127.0.0.1:8879/charts
incubator http://storage.googleapis.com/kubernetes-charts-incubator
```

仓库服务器上的可用Charts及其版本等经常会发生更新，必要时，可使用“helm repo update”命令获取最新的仓库元数据信息：

```
~]$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "incubator" chart repository
Update Complete. *Happy Helming! *
```

而要删除指定的仓库配置，“helm repo remove<REPO_NAME>”即可轻松完成。

4.配置依赖关系

构建存在依赖关系的Charts时，还需要为其定义依赖项，例如，前面创建的myapp依赖于数据库管理系统MySQL时，在mychart的目录

中创建`requirements.yaml`文件给出依赖的Charts列表定义其依赖关系即可，文件内容类似如下所示：

```
dependencies:
- name: mysql
  version: 0.6.0
  repository: https://kubernetes-charts.storage.googleapis.com
```

而后，需要运行“`helm dependency update`”命令为Charts更新依赖关系。更新过程中，Helm会自动生成一个锁定文件`requirements.lock`，以便后续再次获取依赖关系时使用已知的工作版本。运行下面的命令来引入定义的MySQL依赖项时，会自动下载MySQL相关的Charts程序包至`mychart/charts`子目录中，如下面的命令输出结果所示：

```
~]$ helm dependency update ./mychart
Hang tight while we grab the latest from your chart repositories...
.....
Saving 1 charts
Downloading mysql from repo https://kubernetes-charts.storage.googleapis.com
Deleting outdated charts
```

此时，再次部署myapp Charts，就会同时部署依赖到的mysql Charts。另外，用户也可以手动将所依赖到的程序包直接放置于`mychart/charts`目录中来定义依赖关系，此时不必要再使用`requirements.yaml`文件。

15.3 Helm实践：部署EFK日志管理系统

应用程序的日志收集和监控通常是其必要的外围功能，它们有助于记录、分析性能表现及排查故障等，例如此前在查看Pod对象的日志时使用的`kubectl log`命令便是获取容器化应用日志的一种方式。然而，对于分布式部署的应用来说，类似这种逐一查看各实例相关日志的方式存在着操作烦琐且效率低下等诸多问题，再加上需要额外获取操作系统级别的多个日志源中的日志信息，其管理成本势必会进一步上升。解决此类需求的常见方案是使用集中式日志存储和管理系统，它们于各节点部署日志采集代理程序从日志源采集日志并发往中心存储管理系统，并经由单个面板进行数据可视化。事实上，对于任何基础设施或分布式系统，统一日志管理都是必不可少的基础组件。同样地，Kubernetes也要实现在整个集群级别收集和聚合日志，以便用户可以从单个仪表盘监控整个集群，其常用的架构形式之一，如图15-3所示。一种流行的开源解决方案是将`fluentd`作为节点级代理程序进行日志采集，并将之聚合存储于Elasticsearch进行日志分析，以及通过Kibana进行数据可视化。这种组合通常简称为EFK。

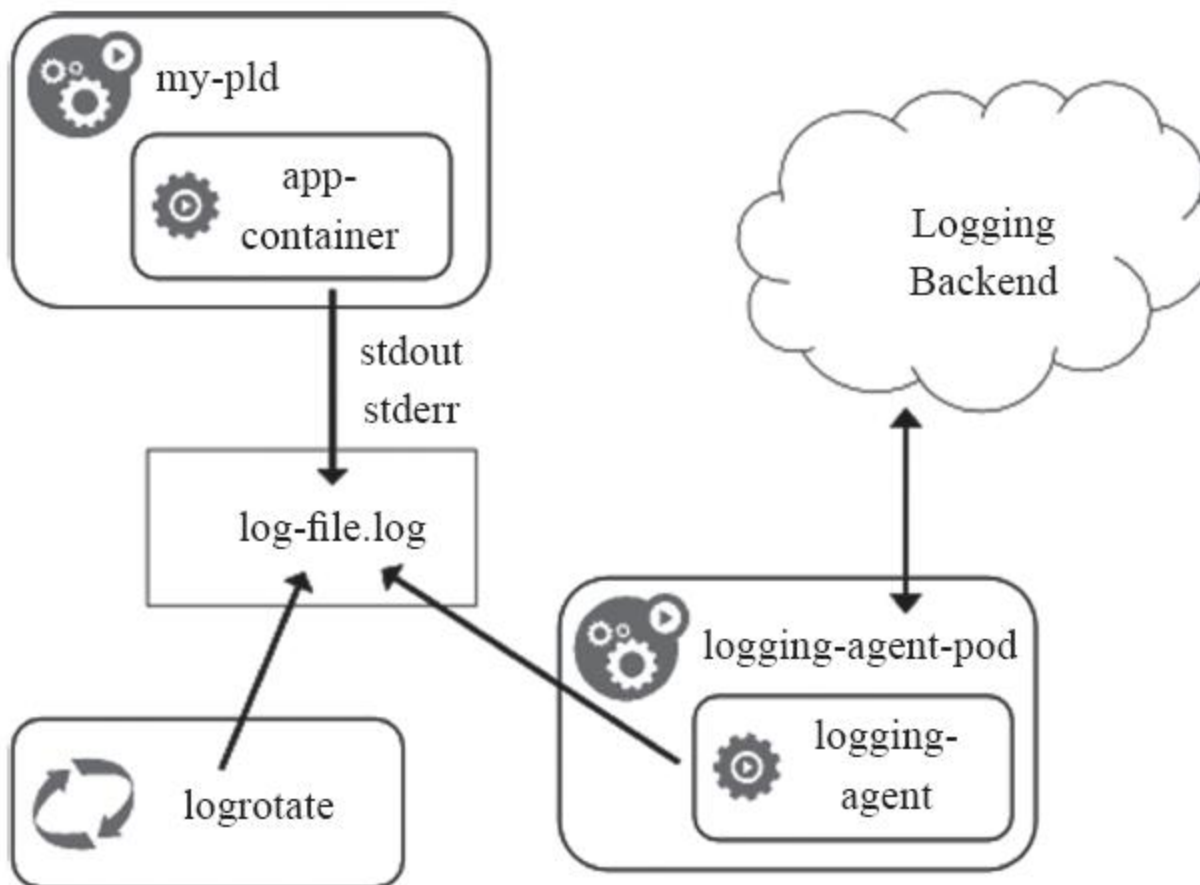


图15-3 基于节点日志采集代理完成日志收集

在Kubernetes上部署fluentd和Kibana的方式易于实现，fluentd由DaemonSet控制器部署于集群中的各节点，而Kibana则由Deployment控制器部署并确保其持续运行即可。但Elastic-Search是一个有状态的应用，需要使用StatefulSet 控制器创建并管理相关的Pod对象，而且它们还分别需要专用的持久存储系统存储日志数据，因此，其部署过程较之前两者要略为烦琐，其部署架构如图15-4所示。

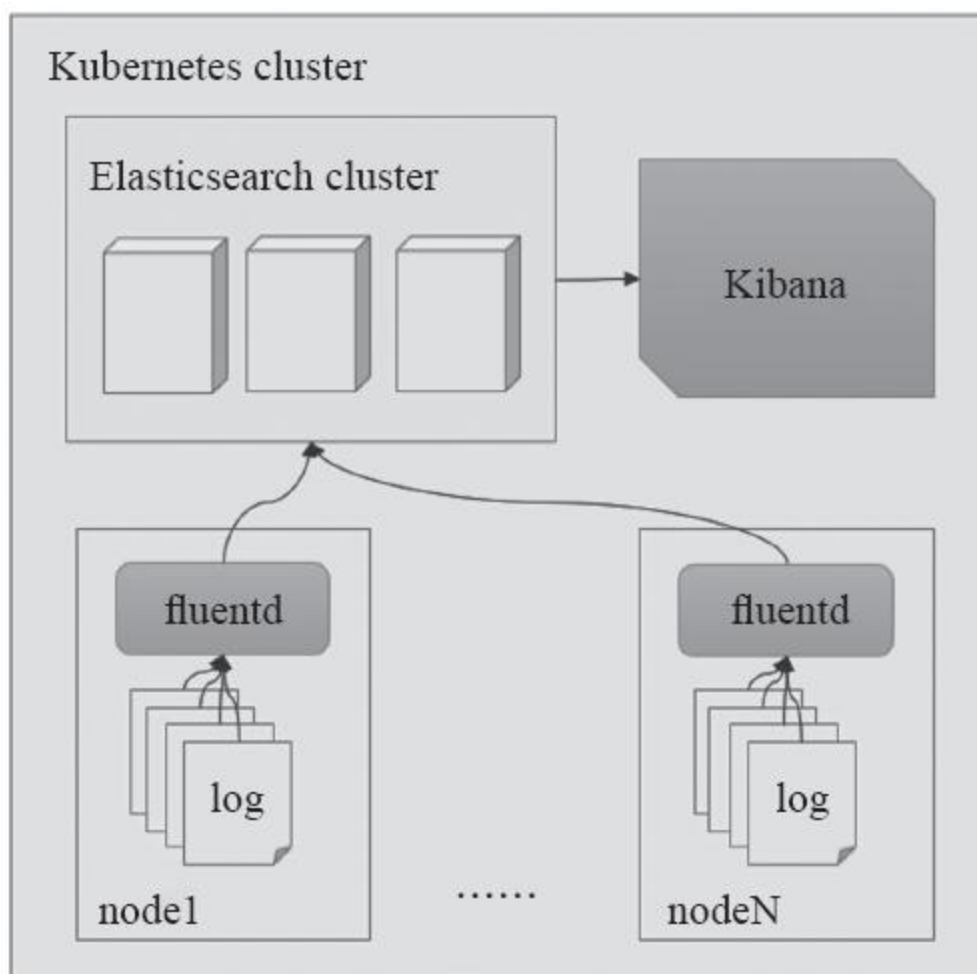


图15-4 Kubernetes集群上的fluentd、Elasticsearch和Kibana

可用的部署方式除了Kubernetes项目在其Addons目录中提供了资源配置清单用于部署EFK之外，Kubeapps (<https://hub.kubeapps.com>) 也为此三者分别提供了相应的Charts定义以帮助用户通过Helm轻松完成其部署。

15.3.1 Elasticsearch集群

ElasticSearch相关Charts位于Kubeapps的incubator仓库中，它使用StatefulSet和Deployment控制器实现了一个可动态伸缩的ElasticSearch集群，并将集群的角色分离为三类节点：客户端节点（上载节点）、master节点和data节点，各自负载实现集群的一部分功能，如图15-5所示。每个master节点和data节点分别需要用到各自的存储卷以持久存储数据，incubator/elasticsearch默认定义它们通过PVC依赖于default或指定的存储类动态创建PV存储卷。

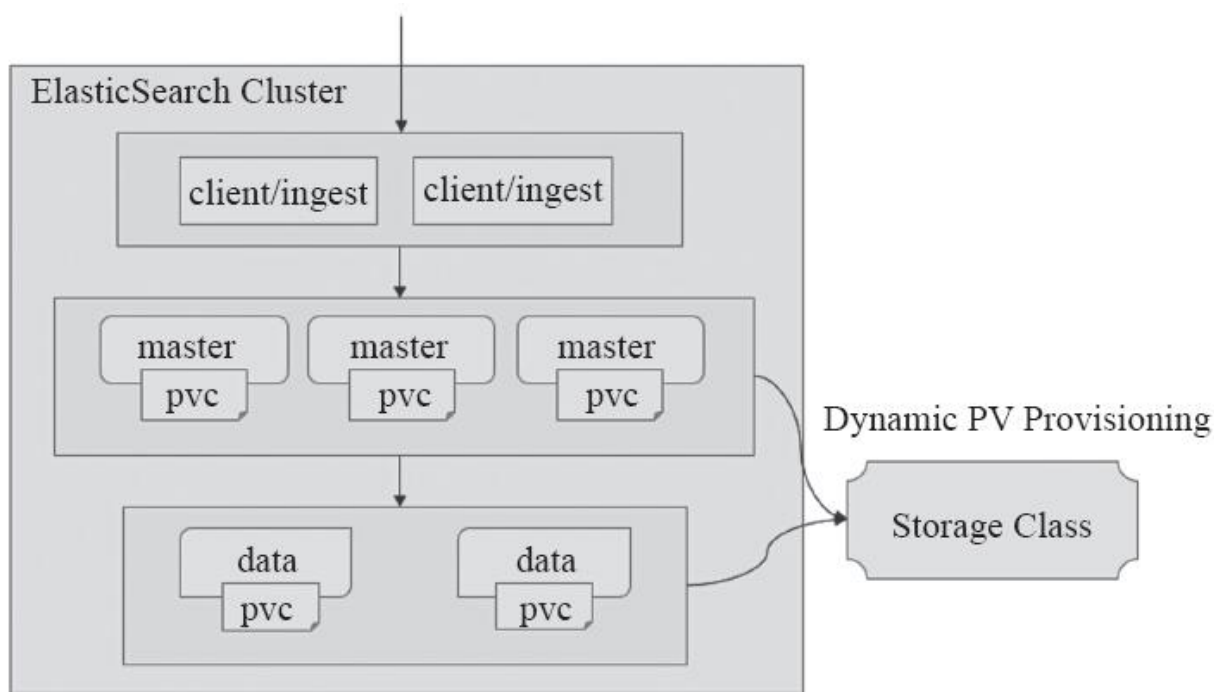


图15-5 多角色ElasticSearch集群

- 客户端节点：也称上载节点或摄取节点，负责执行由一个或多个摄取处理器组成的预处理流水线。需要两个以上客户端实例的场景并不多见，因此其副本数通常为2。

- master节点：负责轻量级群集范围的相关操作，如创建或删除索引、跟踪并判定集群的成员节点，以及决定将哪些分片分配到哪些节点等，因此，一个健康的集群必须要有一个稳定的主节点。master节点

的数量取决于公式“（客户端副本数/2）+1”的计算结果，但至少应该为3。

·数据节点：负责保存包含编入索引的文档的分片，并处理与数据相关的操作，例如数据的增删改查（CRUD）、搜索和聚合等，这些操作通常是I/O、内存和CPU密集型的任务，其所需的节点数量取决于存储及计算的实际需求，因此，监视这些资源并在过载时添加更多的数据节点非常重要。

incubator/elasticsearch的部署模板中定义了众多配置参数，随着Charts版本的迭代，各参数的默认值也可能会有变动，当前的0.4.9版本中，使用的镜像来自仓库“centerforopencscience/elasticsearch”，镜像文件的标签为“5.4”，各master节点的PVC存储卷大小都是4Gi，各data节点的PVC存储卷大小均为30Gi，rbac相关的各资源创建为禁用状态。部署于生产环境时，默认设置中的资源请求和资源限制，以及data节点的PVC存储卷空间等较小，而且在启用了rbac授权插件的集群中还需要创建elasticsearch所需要的各ClusterRole及ClusterRoleBinding资源。这些需要自定义的配置参数可以通过values文件进行设置，或者直接由“helm install”命令的“--set”选项提供。一个示例性的values文件（els-values.yaml）如下所示：

```
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
appVersion: "5.5"

image:
  repository: "centerforopencscience/elasticsearch"
  tag: "5.5"
  pullPolicy: "IfNotPresent"

cluster:
  name: "elasticsearch"
  config:
  env:
    MINIMUM_MASTER_NODES: "2"

client:
  name: client
  replicas: 2
  serviceType: ClusterIP
  heapSize: "2048m"
  antiAffinity: "soft"
  resources:
    limits:
      cpu: "1"
      # memory: "8192Mi"
    requests:
      cpu: "25m"
```

```

        memory: "2048Mi"

master:
  name: master
  exposeHttp: false
  replicas: 3
  heapSize: "2048m"
  persistence:
    enabled: true
    accessMode: ReadWriteOnce
    name: data
    size: "4Gi"
    #storageClass: "glusterfs"
  antiAffinity: "soft"
  resources:
    limits:
      cpu: "1"
      # memory: "8192Mi"
    requests:
      cpu: "25m"
      memory: "2048Mi"

data:
  name: data
  exposeHttp: false
  replicas: 2
  heapSize: "4096m"
  persistence:
    enabled: true
    accessMode: ReadWriteOnce
    name: data
    size: "120Gi"
    #storageClass: "glusterfs"
  terminationGracePeriodSeconds: 3600
  antiAffinity: "soft"
  resources:
    limits:
      cpu: "1"
      # memory: "16384Mi"
    requests:
      cpu: "25m"
      memory: "4096Mi"

## Install Default RBAC roles and bindings
rbac:
  create: true

```

需要特别说明的是，未明确定义持久存储使用的存储类时，无须持久保存数据，或者无可用的实现动态供给PV的存储时，也可以使用emptyDir存储卷，实现方法是将上面示例中master.persistence.enabled和data.persistence.enabled配置参数的值分别设置为“false”。



提示 incubator/elasticsearch相关的配置参数列表获取地址为<https://github.com/kubernetes/charts/tree/master/incubator/elasticsearch>。

下面的命令将在logs名称空间中部署ElasticSearch集群的各角色及其相关的其他资源，其通过前面示例中的Values文件els-values.yaml获取各自定义的配置参数：

```
~]$ kubectl create namespace logs
~]$ helm install incubator/elasticsearch -n efk-els --namespace=logs -f els-values.yaml
```



提示 仅用于测试，且无可用的实现动态供给PV的存储类时，则无须定义Values文件，并将上面的命令替换为“helm install incubator/elasticsearch -n efk-els --namespace=logs --set master.persistence.enabled="false", data.persistence.enabled="false", rbac.create="true"”即可。

从命令最后提供的类似如下的提示信息中可以看出，部署完成后在Kubernetes集群内部访问ElasticSearch服务的接入端点为“efk-elasticsearch-client.logs.svc.cluster.local: 9200”。随后部署fluentd时将基于此访问端点将采集到的日志导入到ElasticSearch集群中，而Kibana也将通过此端点为ElasticSearch中的数据提供可视化的搜索及展示接口：

```
NOTES:
The elasticsearch cluster has been installed.

Elasticsearch can be accessed:

* Within your cluster, at the following DNS name at port 9200:

  efk-els-elasticsearch-client.logs.svc.cluster.local

.....
```

使用cirros镜像创建一个测试客户端，通过此端点对ElasticSearch的服务发起测试访问请求。例如，下面的命令可用于请求显示ElasticSearch集群的欢迎信息：

```
~]$ kubectl run cirros-$RANDOM --rm -it --image=cirros -- sh
/ # curl http://efk-els-elasticsearch-client.logs.svc.cluster.local:9200
{
  "name" : "efk-els-elasticsearch-client-55dccf5f75-7hhc9",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : " uGffeD5bSo08q29v4uLaBw",
  "version" : {
```

```
    "number" : "5.5.2",
    "build_hash" : "b2f0c09",
    "build_date" : "2017-08-14T12:33:14.154Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.0"
  },
  "tagline" : "You Know, for Search"
}
/ #
```

还可以通过类似如下的命令了解ElasticSearch集群当前的工作状态，它显示出当前集群处于正常工作状态“green”，共有7个节点，其中有2个节点为数据节点。由于是新创建的集群，因此目前尚不存在任何数据分片：

```
/ # curl http://efk-els-elasticsearch-client.logs.svc.cluster.local:9200/_cluster/healthpretty
{
  "cluster_name" : "elasticsearch",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 7,
  "number_of_data_nodes" : 2,
  "active_primary_shards" : 0,
  "active_shards" : 0,
  .....
}
/ #
```

15.3.2 日志采集代理fluentd

fluentd是一个开源的数据收集器，基于C和Ruby语言开发，它目前有数百种以Ruby Gem形式独立存在的可选插件，用于连接多种数据源和数据输出组件等，如fluent-plugin-elasticsearch插件用于实现将采集到的数据发送给ElasticSearch。



提示 fluentd的可用插件列表获取地址为<https://www.fluentd.org/plugins>。

运行时，fluentd从配置文件获取数据源、数据输出目标、过滤器等相关的配置信息，这些配置信息以source、match、filter、system、label和@include配置参数给出。

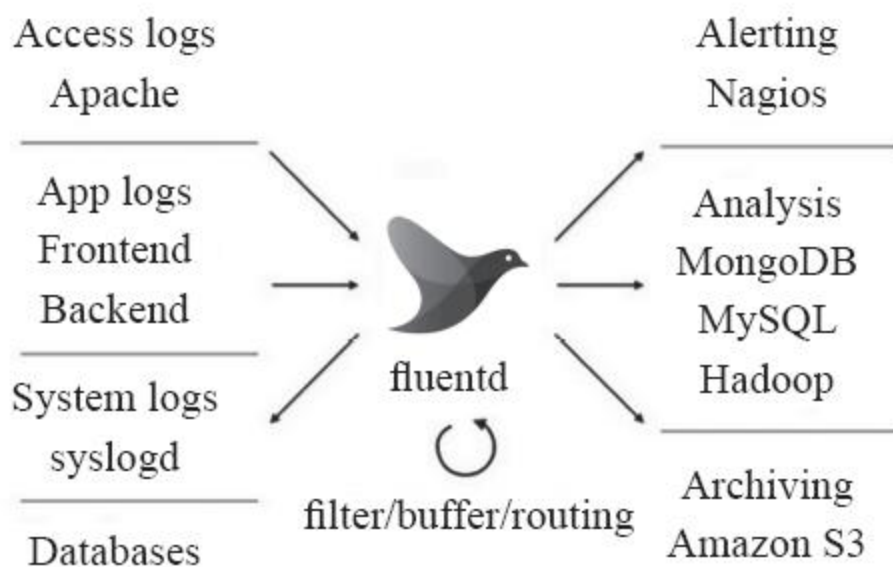


图15-6 fluentd架构图

- source: 定义数据源，每个数据源都需要有其专有类型的定义。
- match: 数据输出的目标位置，如文件或各种存储系统。

·**filter**: 过滤器，即事件处理流水线，通常运行于输入和输出之间。

·**system**: 系统级设置，如处理器名称等。

·**label**: 用于分组过滤器及输出目标。

·**@include**: 引用配置信息中某些已有的定义，以达到配置复用之目的。

下面的配置示例中给出了fluentd收集Kubernetes平台上容器日志的输入及输出相关的定义：

```
<source>
  @id fluentd-containers.log
  @type tail
  path /var/log/containers/*.log
  pos_file /var/log/fluentd-containers.log.pos
  time_format %Y-%m-%dT%H:%M:%S.%NZ
  tag raw.kubernetes.*
  format json
  read_from_head true
</source>
<match raw.kubernetes.**>
  @id raw.kubernetes
  @type detect_exceptions
  remove_tag_prefix raw
  message log
  stream stream
  multiline_flush_interval 5
  max_bytes 500000
  max_lines 1000
</match>
```

incubator/fluentd-elasticsearch是Kubeapps上定义的基于fluentd收集容器及系统日志并将其输出至ElasticSearch中的Charts之一，它内置了基于ConfigMap资源定义的fluentd配置文件，基本无须修改即能投入使用。另外，fluentd是运行于各节点上的日志采集代理，因此，它受控于DaemonSet控制器。

基于此Charts部署fluentd时通常仅需为其指定ElasticSearch服务的访问端点即可，因此可直接基于命令启动部署操作：

```
~]$ helm install local/fluentd-elasticsearch -n efk-flu --namespace=logs \
  --set elasticsearch.host="efk-els-elasticsearch-
```

```
client.logs.svc.cluster.local"
```

不过，若需要收集master节点上的日志，就需要为部署的Pod对象添加tolerations以容忍master上的taints。修改Charts中默认的values.yaml文件中的配置，并以之完成部署即能实现。

确认fluentd相关的各Pod对象正常运行之后，即可到ElasticSearch集群中查看其收集并存储的日志的索引及数据信息，例如，仍于此前启动的测试Pod中向ElasticSearch发起访问请求，查看已生成的索引，命令如下：

```
/ # curl http://efk-els-elasticsearch-
client.logs.svc.cluster.local:9200/_cat/indices
green open logstash-2018.06.10 UJtHdGeHTISVfKw8s0_PjQ 5 1 6 0 337.3kb 168.6kb
green open logstash-2018.06.09 eotV9S2QQuCK1JUPEnmieQ 5 1 47470 0 97.6mb 48.9mb
/ #
```

命令结果中显示出以“logstash-YYYY.MM.DD”格式命名的索引列表，即表示fluentd已经能够正常采集到日志数据并输出到指定的ElasticSearch集群中。

15.3.3 可视化组件Kibana

Kibana是ElasticSearch的数据分析及可视化平台，能够用来搜索、查看存储在Elastic-Search索引中的数据。它可以通过各种图表进行高级数据分析及展示，用户基于Web GUI可以快速创建仪表盘

（dashboard）实时显示ElasticSearch的查询结果。Kibana配置过程简单便捷，图形样式丰富，可借助于ElasticSearch创建柱形图、折线图、散点图、直方图、饼图和地图等数据展示接口。Kibana增强了ElasticSearch的数据分析能力，让用户能够更加智能地分析数据和展示数据。

类似于fluentd，Kibana也通过URL访问ElasticSearch，但它要通过环境变量ELASTIC-SEARCH_URL来指定。部署于Kubernetes上的Kibana一般会由集群外的客户端访问，因此需要为其配置Ingress资源，也可以使用NodePort或LoadBalancer类型的Service资源进行服务暴露。它默认通过HTTP提供服务，在使用Ingress暴露到互联网时，建议将其配置为HTTPS类型的服务。

stable/kibana是Kubeapps上提供的Charts，下面是适用于当前配置环境设定的Values文件（kibana-values.yaml），它使用镜像文件kibana: 5.5运行容器，环境变量ELASTICSEARCH_URL的值为前面部署的ElasticSearch集群的访问接口，并通过NodePort类型的Service资源进行服务暴露：

```
image:
  repository: "kibana"
  tag: "5.5"
  pullPolicy: "IfNotPresent"

env:
  ELASTICSEARCH_URL: http://efk-els-elasticsearch-client.logs.svc:9200
  SERVER_PORT: 5601

service:
  type: NodePort
  externalPort: 443
  internalPort: 5601

ingress:
  enabled: false
```

而后，使用下面的命令即可完成Kibana部署：

```
-]$ helm install stable/kibana -n efk-kib --namespace=logs -f kibana-values.yaml
NAME:      efk-kib
LAST DEPLOYED: Sun Jun 10 09:19:49 2018
...

RESOURCES:
==> v1/Service
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
efk-kib-kibana      NodePort    10.110.201.6  <none>       443:30978/TCP  0s
.....
```

确认相关的Pod资源能够正常运行之后，即可通过如上面命令显示的Service资源中30978端口于集群外访问Kibana，其初始界面如图15-7所示。在“Management”中添加相应的索引模式加载相关的索引数据即可完成数据搜索及可视化配置，由fluentd收集的日志以“logstash-YYYY.MM.DD”为索引名根据索引产生的日志按天保存于单独的索引中，因此Kibana默认的模式“logstash-*”便能够加载这些索引中的数据。

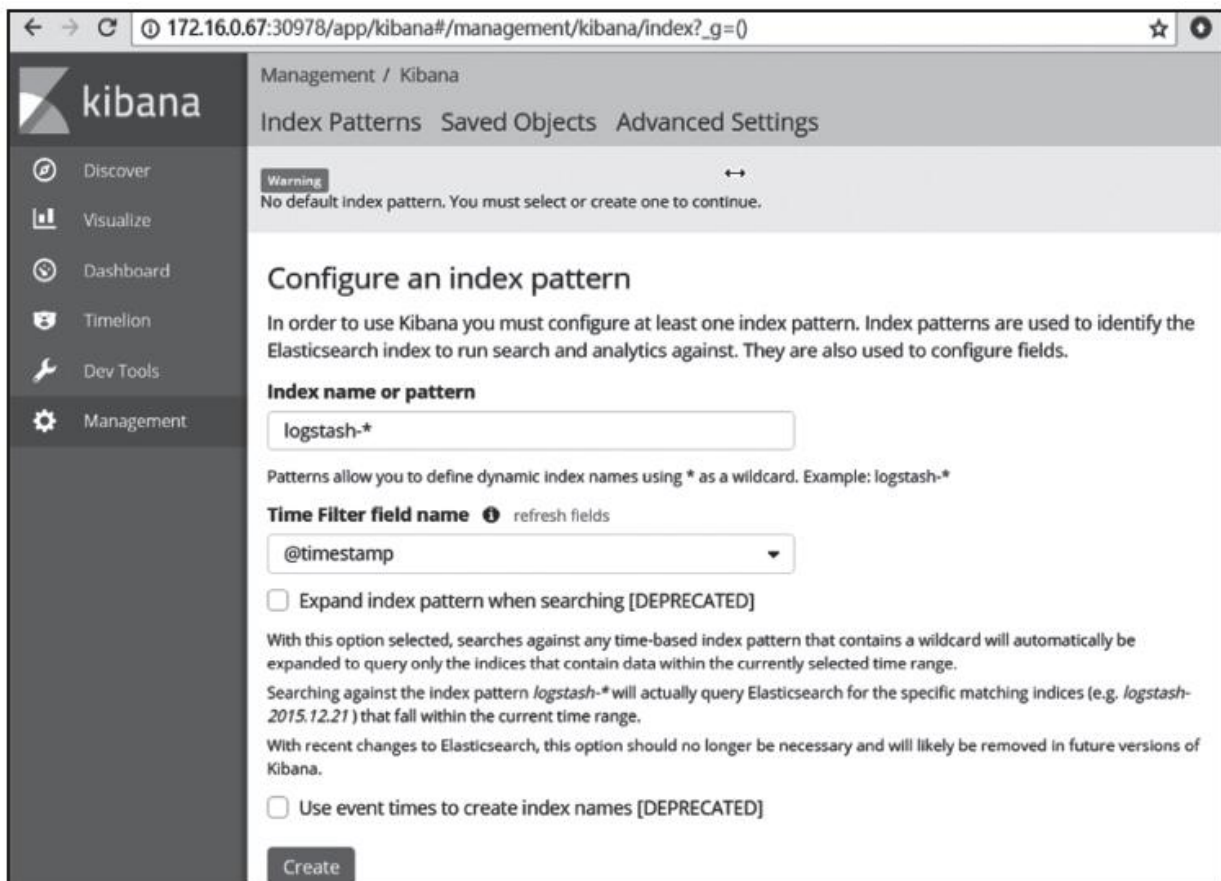


图15-7 Kibana的初始界面

创建好索引模式之后，即可通过“**Discover**”搜索数据，或者在“**Visualize**”界面中定义可视化图形，并将它们集成于可在“**Dashboard**”中创建的仪表板里。



注意 Kibana程序（非Charts）的版本号不能高于ElasticSearch，否则会导致不兼容。

15.4 本章小结

目前来说，Kubernetes部署及管理应用程序的接口仍然相当复杂，在维护较多的资源时，用户必然会受困于其复杂多变的资源配置清单，Helm通过Charts实现了类似于yum、dnf或apt-get等程序包管理器的功能，大大降低了用户的使用成本。本章详细讲解了Helm的使用方式，并通过示例演示了其使用方法。

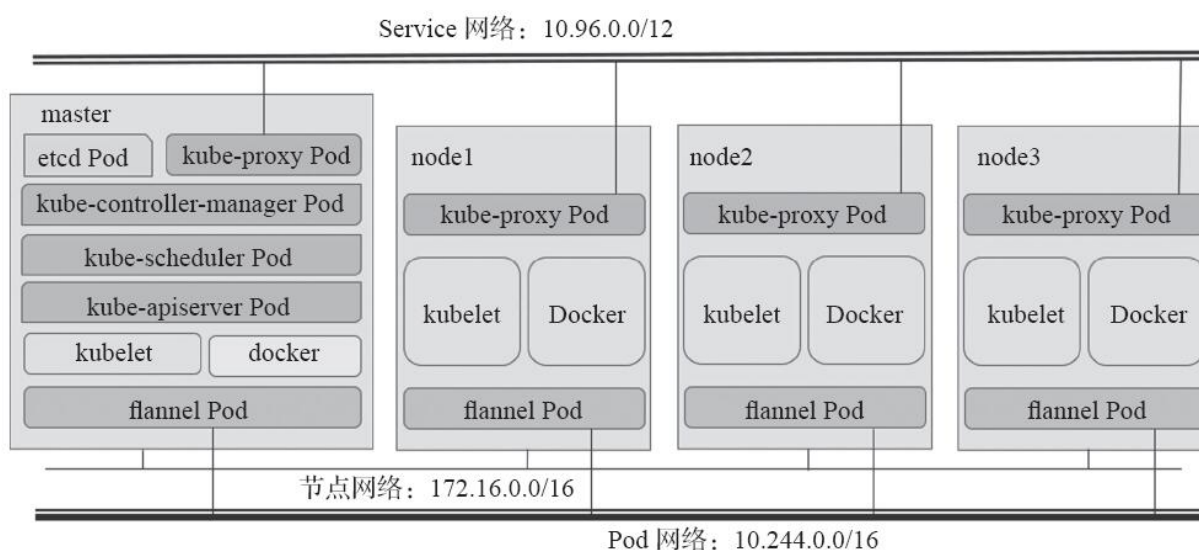
附录A 部署Kubernetes集群

A.1 准备部署Kubernetes集群

Kubernetes项目目前仍然处于快速迭代阶段，演示过程中使用的配置对于其后续版本可能存在某些变动，因此，版本不同时，对具体特性支持的变动请读者参考Kubernetes的ChangeLog或其他相关文档中的说明。

A.1.1 部署目标

图A-1给出了本节要部署的目标集群的基本环境，它拥有一个Master主机和三个Node主机。各Node主机的配置方式基本相同。



图A-1 Kubernetes集群部署目标示意图

各主机上采用的容器运行时环境为Docker，为Pod提供网络功能的CNI是flannel，它运行在托管于Kubernetes之上的Pod对象，另外，基础附件还包括KubeDNS（或CoreDNS）用于名称解析和服务发现。

A.1.2 系统环境及部署准备

如前所述，Kubernetes当前仍处于快速迭代的周期中，其版本变化频繁、跨版本的特性变化较大，为了帮助读者确认各配置和功能的可用性，本书使用如下基础环境。

1.各相关组件及主机环境

操作系统、容器引擎、etcd及Kubernetes的相关版本分别如下。

- OS: Cent OS 7.5x86_64
- Container runtime: Docker 18.06.ce
- Kubernetes: 1.12

各主机角色分配及IP地址如表A-1所示。

表A-1 Kubernetes集群的主机环境

IP 地址	主机名	角色
172.16.0.70	master, master.ilinux.io	master
172.16.0.66	node01, node01.ilinux.io	node
172.16.0.67	node02, node02.ilinux.io	node
172.16.0.68	node03, node03.ilinux.io	node

2.基础环境设置

Kubernetes的正确运行依赖于一些基础环境的设定，如各节点时间通过网络时间服务保持同步和主机名称解析等，集群规模较大的实践场景中，主机名称解析通常由DNS服务器完成。本测试示例中，时间同步服务直接基于系统的默认配置从互联网的时间服务中获取，主机名称解析则由hosts文件进行。

(1) 主机名称解析

分布式系统环境中的多主机通信通常基于主机名称进行，这在IP地址存在变化的可能性时为主机提供了固定的访问入口，因此一般需要有专用的DNS服务负责解决各节点主机名。不过，考虑到此处部署的是测试集群，因此为了降低系统的复杂度，这里将采用基于hosts的文件进行主机名称解析。编辑Master和各Node上的/etc/hosts文件，确保其内容如下：

```
172.16.0.66      node01.ilinux.io node01
172.16.0.67      node02.ilinux.io node02
172.16.0.68      node03.ilinux.io node03
172.16.0.70      master.ilinux.io master
```

(2) 主机时间同步

如果各主机可直接访问互联网，则直接启动各主机上的`chronyd`服务即可。否则需要使用本地网络中的时间服务器，例如，可以将Master配置为`chrony server`，而后其他节点均从Master同步时间：

```
~]# systemctl start chronyd.service
~]# systemctl enable chronyd.service
```

(3) 关闭防火墙服务

各Node运行的`kube-proxy`组件均要借助`iptables`或`ipvs`构建Service资源对象，该资源对象是Kubernetes的核心资源之一。出于简化问题复杂度之需，这里需要事先关闭所有主机之上的`iptables`或`firewalld`服务：

```
~]# systemctl stop firewalld.service iptables.service
~]# systemctl disable firewalld.service
~]# systemctl disable iptables.service
```

(4) 关闭并禁用SELinux

若当前启用了SELinux，则需要临时设置其当前状态为`permissive`：

```
~]# setenforce 0
```

另外，编辑`/etc/sysconfig/selinux`文件，以彻底禁用SELinux：

```
~]# sed -i 's@^\(SELINUX=\).*@1disabled@' /etc/sysconfig/selinux
```

(5) 禁用Swap设备（可选步骤）

kubeadm默认会预先检查当前主机是否禁用了Swap设备，并在未禁用时强制终止部署过程。因此，在主机内存资源充裕的条件下，需要禁用所有的Swap设备。

关闭Swap设备，需要分两步完成。首先是关闭当前已启用的所有Swap设备：

```
~]#swapoff -a
```

而后编辑/etc/fstab配置文件，注释用于挂载Swap设备的所有行。不同系统环境默认启用的Swap设备不尽相同，请读者根据实际情况完成相应操作。另外，部署时也可以选不禁用Swap，而是通过后文的kubeadm init及kubeadm join命令执行时额外使用相关的选项忽略检查错误。

(6) 启用ipvs内核模块（可选步骤）

Kubernetes 1.11之后的版本默认支持使用ipvs代理模式的Service资源，但它依赖于ipvs相关的内核模块，而这些模块默认不会自动载入。因此，这里选择创建载入内核模块相关的脚本文件/etc/sysconfig/modules/ipvs.modules，设定于系统引导时自动载入的ipvs相关的内核模块，以支持使用ipvs代理模式的Service资源。文件内容如下：

```
#!/bin/bash
ipvs_mods_dir="/usr/lib/modules/$(uname -r)/kernel/net/netfilter/ipvs"
for i in $(ls $ipvs_mods_dir | grep -o "^[^.]*"); do
    /sbin/modinfo -F filename $i &> /dev/null
    if [ $? -eq 0 ]; then
        /sbin/modprobe $i
    fi
done
```

而后，修改文件权限，并手动为当前系统环境加载内核模块：

```
~]# chmod +x /etc/sysconfig/modules/ipvs.modules
~]# /etc/sysconfig/modules/ipvs.modules
```

不过，ipvs仅负责实现负载均衡相关的任务，它无法完成kube-proxy中的包过滤及SNAT等功能，这些仍需要由iptables实现。另外，对于初学者来说，前期的测试并非必然要用到ipvs代理模式，部署时可省略此步骤。

A.2 部署Kubernetes集群

kubeadm是用于快速构建Kubernetes集群的工具，随着Kubernetes的发行版本而提供，使用它构建集群时，大致可分为如下几步。

- 1) 在Master及各Node安装Docker、kubelet及kubeadm，并以系统守护进程的方式启动Docker和kubelet服务。
- 2) 在Master节点上通过kubeadm init命令进行集群初始化。
- 3) 各Node通过kubeadm join命令加入初始化完成的集群中。
- 4) 在集群上部署网络附件，如flannel或Calico等以提供Service网络及Pod网络。

为了简化部署过程，kubeadm使用一组固定的目录及文件路径存储相关的配置及数据文件，其中/etc/kubernetes目录是所有文件或目录的统一存储目录。它使用/etc/kubernetes/manifests目录存储各静态Pod资源的配置清单，用到的文件有etcd.yaml、kube-apiserver.yaml、kube-controller-manager.yaml和kube-scheduler.yaml四个，它们的作用基本能够见名知义。另外，/etc/kubernetes/目录中还会为Kubernetes的多个组件存储专用的kubeconfig文件，如kubelet.conf、controller-manager.conf、scheduler.conf和admin.conf等，它们分别为相关的组件提供接入API Server的认证信息等。此外，它还会在/etc/kubernetes/pki目录中存储若干私钥和证书文件。

A.2.1 设定容器运行环境

Kubernetes支持多种容器运行时环境，例如Docker、RKT和Frakti等，本书将采用其中目前最为流行的、接受程度最为广泛的Docker，它的常用部署方式有两种，具体如下。

·由系统发行版的程序包仓库提供，如Cent OS 7Extras仓库中的Docker。

·Docker官方仓库中的程序包，以Cent OS 7为例，它通常能够提供较Extras仓库中更新版本的程序包，获取地址为<https://download.docker.com/>。

本文采用的是第二种方式，不过，Kubernetes认证的Docker版本通常略低于其最新版本，因此生产环境部署时应该尽可能部署经过认证的版本。考虑到Kubernetes版本迭代周期较短，它对Docker版本的支持也会快速变化，因此本示例将直接使用Docker仓库中的最新版本。部署时，Docker需要安装于Master及各Node主机之上，安装方式相同，其步骤如下：

```
~]# wget https://download.docker.com/linux/centos/docker-ce.repo \
    -O /etc/yum.repos.d/docker-ce.repo
~]# yum install docker-ce
```



注意 kubeadm构建集群的过程需要到gcr.io中获取Docker镜像，因此必须确保Docker主机能够正常访问到此站点，否则，就得配置Docker以代理的方式访问gcr.io，或者配置kubeadm从其他Registry获取相关的镜像。代理的方法是在[service]配置段中添加类似如下格式的配置项：Environment="HTTP_PROXY=http://IP: PORT"，或Environment="HTTPS_PROXY=https://IP: PORT"。

另外，Docker自1.13版起会自动设置iptables的FORWARD默认策略为DROP，这可能会影响Kubernetes集群依赖的报文转发功能，因此，需要在docker服务启动后，重新将FORWARD链的默认策略设置为ACCEPT，方式是修改/usr/lib/systemd/system/docker.service文件，在“ExecStart=/usr/bin/dockerd”一行之后新增一行如下内容：

```
ExecStartPost=/usr/sbin/iptables -P FORWARD ACCEPT
```

上面各步骤设置完成后即可启动docker服务，并将其设置为随系统引导而自动启用，相关命令如下：

```
~]# systemctl daemon-reload
~]# systemctl start docker.service
~]# systemctl enable docker.service
```



提示 国内访问DockerHub下载镜像的速度较缓慢，建议使用国内的镜像对其进行加速，如<https://registry.docker-cn.com>，另外，中国科技大学也提供了公共可用的镜像加速服务，其URL为<https://docker.mirrors.ustc.edu.cn>，将其定义在daemon.json中重启Docker即可使用。

A.2.2 设定Kubernetes集群节点

kubelet是运行于集群中每个节点之上的Kubernetes代理程序，它的核心功能在于通过API Server获取调度至自身运行的Pod资源的PodSpec并依之运行Pod对象。事实上，以自托管方式部署的Kubernetes集群，除了kubelet和Docker之外的所有组件均以Pod对象的形式运行。

1. 安装kubelet及kubeadm

安装kubelet的常用方式包含如下几种。快速迭代期内，Linux发行商提供的安装包通常版本较低，因此建议采用下列方式的第一种或第二种，本章将采用第二种方式。

- Kubernetes提供的二进制格式的tar包。
- Google的yum仓库中提供的rpm包，可通过国内的镜像站点获取，例如阿里云镜像站。
- OS发行商提供的安装包，例如Cent OS 7Extras仓库中的Kubernetes相关程序包。



提示 对于rpm方式的安装来说，kubelet、kubeadm和kubectl等是各自独立的程序包，Master及各Node至少应该安装kubelet和kubeadm，而kubectl则只需要安装于客户端主机即可，不过，由于依赖关系它通常也会被自动安装。另外，Google提供的kubelet rpm包的yum仓库托管于Google站点的服务器主机之上，目前访问起来略有不便。

幸运的是，目前国内的阿里云等镜像（<http://mirrors.aliyun.com>）对此项目也有镜像提供。

首先设定用于安装kubernetes、kubeadm和kubectl等组件的yum仓库，编辑配置文件/etc/yum.repos.d/kubernetes.repo，内容如下：

```
[kubernetes]
name=Kubernetes
baseurl= https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64/
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey= https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg https://
mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
```

而后执行如下命令即可安装相关的程序包：

```
[root@master ~]# yum install kubelet kubeadm kubectl
```

2.配置kubelet

Kubernetes自1.8版本起强制要求关闭系统上的交换分区（Swap），否则kubelet将无法启动。当然，用户也可以通过将kubelet的启动参数“--fail-swap-on”设置为“false”忽略此限制，尤其是系统上运行有其他重要进程且系统内存资源稍嫌不足时建议保留交换分区。

编辑kubelet的配置文件/etc/sysconfig/kubelet，设置其配置参数如下，以忽略禁止使用Swap的限制：

```
KUBELET_EXTRA_ARGS="--fail-swap-on=false"
```

待配置文件修改完成后，需要设定kubelet服务开机自动启动，这也是kubeadm的强制要求：

```
[root@master~]#systemctl enable kubelet.service
```

A.2.3 集群初始化

一旦Master和各Node的Docker及kubelet设置完成后，接着便可以在Master节点上执行“`kubeadm init`”命令进行集群初始化。`kubeadm init`命令支持两种初始化方式，一是通过命令行选项传递关键的参数设定，另一个是基于yaml格式的专用配置文件设定更详细的配置参数。下面分别给出了两种实现方式的配置步骤，建议读者采用第二种方式。

Master初始化方式一： 运行下面的命令，便可完成Master的初始化。

```
[root@master ~]# kubeadm init \
--kubernetes-version=v1.12.1 \
--pod-network-cidr=10.244.0.0/16 \
--service-cidr=10.96.0.0/12 \
--apiserver-advertise-address=0.0.0.0 \
--ignore-preflight-errors=Swap
```

上面命令的选项及参数设定决定了集群运行环境的众多特性设定，这些设定对于此后在集群中部署运行应用程序至关重要。

·`--kubernetes-version`: 正在使用的Kubernetes程序组件的版本号，需要与kubelet的版本号相同。

·`--pod-network-cidr`: Pod网络的地址范围，其值为CIDR格式的网络地址；使用flannel网络插件时，其默认地址为10.244.0.0/16。

·`--service-cidr`: Service的网络地址范围，其值为CIDR格式的网络地址，默认地址为10.96.0.0/12。

·`--apiserver-advertise-address`: API server通告给其他组件的IP地址，一般应该为Master节点的IP地址，0.0.0.0表示节点上所有可用的地址。

·`--ignore-preflight-errors`: 忽略哪些运行时的错误信息，其值为Swap时，表示忽略因swap未关闭而导致的错误。



提示 更多的参数请参考kubeadm的文档，链接地址为<https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/>。

Master初始化方式二： `kubeadm init`也可通过配置文件加载配置，以定制更丰富的部署选项。以下是符合前述命令设定方式的使用示例，不过，它明确定义了`kubeProxy`的模式为`ipvs`，并支持通过修改`imageRepository`的值来修改获取系统镜像时使用的镜像仓库。

```
apiVersion: kubeadm.k8s.io/v1alpha2
kind: MasterConfiguration
kubernetesVersion: v1.12.1
api:
  advertiseAddress: 172.20.0.70
  bindPort: 6443
  controlPlaneEndpoint: ""
imageRepository: k8s.gcr.io
kubeProxy:
  config:
    mode: "ipvs"
    ipvs:
      ExcludeCIDRs: null
      minSyncPeriod: 0s
      scheduler: ""
      syncPeriod: 30s
kubeletConfiguration:
  baseConfig:
    cgroupDriver: cgroupfs
    clusterDNS:
      - 10.96.0.10
    clusterDomain: cluster.local
    failSwapOn: false
    resolvConf: /etc/resolv.conf
    staticPodPath: /etc/kubernetes/manifests
networking:
  dnsDomain: cluster.local
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.96.0.0/12
```

将上面的内容保存于配置文件中，如`kubeadm-config.yaml`，而后执行相应的命令即可完成**Master**初始化：

```
~]# kubeadm init --config kubeadm-config.yaml --ignore-preflight-errors=Swap
```

无论使用上述哪种方法，命令的执行过程都会执行众多部署操作并生成相关的信息，如生成配置文件，标识主节点、生成**bootstrap token**及部署核心附加组件**kube-proxy**和**kube-dns**等，尤其是为集群通信安全于`/etc/kubernetes/pki`目录中生成的一众密钥和数字证书，并在最后给出了**kubectl**客户端工具的配置文件生成方式以及随后将**Node**加入集群时使用的引导认证令牌（**bootstrap token**）等，后续需要加入集群的各**Node**都将使用该引导认证令牌加入集群。不同版本的**kubeadm**其输出

结果或许略有不同。本示例特定将需要注意的部分以粗体格式予以标识，它们是后续步骤的重要提示信息。命令执行的结果如下所示：

```
.....
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

    mkdir -p $HOME/.kube
    sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
    https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

    kubeadm join 172.16.0.70:6443 --token sgm8ht.dxhqol0o42i52fnz --discovery-
token-ca-
    cert-hash
    sha256:ffc0304409012dacb07edbe886e36b4215cff94fa5cd48eee2f55c91e65b0a01
```

根据上述输出信息的提示（粗体部分），完成集群部署还需要执行三类操作：设定**kubectl**的配置文件、部署网络附件以及将各**Node**加入集群。下面就来讲解如何进行这三步操作。

A.2.4 设定**kubectl**的配置文件

kubectl是执行**Kubernetes**集群管理的核心工具。默认情况下，**kubectl**会从当前用户主目录（保存于环境变量**HOME**中的值）中的隐藏目录**.kube**下名为**config**的配置文件中读取配置信息，包括要接入**Kubernetes**集群、以及用于集群认证的证书或令牌等信息。集群初始化时，**kubeadm**会自动生成一个用于此类功能的配置文件**/etc/kubernetes/admin.conf**，将它复制为用户的**\$HOME/.kube/config**文件即可直接使用。这里以**Master**节点上的**root**用户为例进行操作，不过，在实践中应该以普通用户的身份进行：

```
[root@master ~]# mkdir -p $HOME/.kube
[root@master ~]# cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

至此为止，一个Kubernetes Master节点已经基本配置完成。接下来即可通过API Server来验证其各组件的运行是否正常。kubectl有着众多子命令，其中“get compontsstatuses”即能显示出集群组件当前的状态，也可使用其简写格式“get cs”：

```
[root@master ~]# kubectl get cs
```

NAME	STATUS	MESSAGE	ERROR
scheduler	Healthy	ok	
controller-manager	Healthy	ok	
etcd-0	Healthy	{ "health": "true" }	

若上面命令结果的STATUS字段为“Healthy”，则表示组件处于健康运行状态，否则需要检查其错误所在，必要时可使用“kubeadm reset”命令重置之后重新进行集群初始化。

另外，使用“kubectl get nodes”命令能够获取集群节点的相关状态信息，如下命令结果显示了Master节点的状态为“NotReady”（未就绪），这是因为集群中尚未安装网络插件所致，执行完后面的其他步骤后它即自行转为“Ready”：

```
[root@master ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master.ilinux.io	NotReady	master	4m	v1.12.1

A.2.5 部署网络插件

为Kubernetes提供Pod网络的插件有很多，目前最为流行的是flannel和Calico。相比较来说，flannel以其简单、易部署、易用等特性广受用户欢迎。基于kubeadm部署时，flannel同样运行于Kubernetes集群的附件，以Pod的形式部署运行于每个集群节点上以接受Kubernetes集群管理。事实上，也可以直接将flannel程序包安装并以守护进程的方式运行于集群节点上，即以非托管的方式运行。部署方式既可以是获取其资源配置清单于本地而后部署于集群中，也可以直接在线进行应用部署。部署命令是“kubectl apply”或“kubectl create”，例如，下面的命令将直接使用在线的配置清单进行flannel部署：

```
[root@master ~]# kubectl apply -f  
https://raw.githubusercontent.com/coreos/flannel/
```

kubectl可根据定义资源对象的清单文件将其提交给API Server以管理资源对象，如使用“**kubectl apply -f/PATH/TO/MANIFEST**”命令即可根据清单设置资源的目标状态。**kubectl**的具体使用会在后面的章节进行详细介绍。

配置flannel网络插件时，Master节点上的Docker首先会去获取flannel的镜像文件，而后根据镜像文件启动相应的Pod对象。待其运行完成后再次查看集群中的节点状态可以看出Master已经变为“Ready”状态：

```
[root@master ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master.ilinux.io	Ready	master	10m	v1.12.1



提示 **kube-system|grep flannel**”命令的结果显示Pod的状态为Running时即表示网络插件flannel部署完成。

A.2.6 添加Node至集群中

Master各组件运行正常后即可将各Node添加至集群中。配置节点时，需要事先参考前面“设置容器运行环境”和“设定Kubernetes集群节点”两节中的配置过程设置好Node主机，而后即可在Node主机上使用“**kubeadm join**”命令将其加入集群中。不过，为了系统安全起见，任何一个试图加入到集群中的节点都需要先经由API Server完成认证，其认证方法和认证信息在Master上运行的“**kubeadm init**”命令执行初始化时将于输出结果信息的最后一部分中提供。

例如，类似如下命令即可将node01.ilinux.io加入集群中，它使用集群初始化时生成的认证令牌（token）信息进行认证。另外，同样出于为操作系统及其他应用保留交换分区之目的，在**kubeadm join**命令上添加了“**--ignore-preflight-errors=Swap**”选项：

```
[root@node01 ~]# kubeadm join 172.16.0.70:6443 --ignore-preflight-errors=Swap
--token sgm8ht.dxhqol0o42i52fnz --discovery-token-ca-cert-hash \
```

```
sha256:ffc0304409012dacb07edbe886e36b4215cff94fa5cd48eee2f55c91e65b0a01
```

提供给API Server的bootstrap token认证完成后，`kubeadm join`命令会为后续Master与Node组件间的双向ssl/tls认证生成私钥及证书签署请求，并由Node在首次加入集群时提交给Master端的CA进行签署。默认情况下，`kubeadm`配置`kube-apiserver`启用了bootstrapTLS功能，并支持证书的自动签署。于是，`kubelet`及`kube-proxy`等组件的相关私钥和证书文件在命令执行结束后便可自动生成，它们默认保存于`/var/lib/kubelet/pki`目录中。

在每个节点上重复上述步骤就能够将其加入集群中。所有节点加入完成后，即可使用“`kubectl get nodes`”命令验证集群的节点状态，包括各节点的名称、状态就绪与否、角色（是否为节点Master）、加入集群的时长以及程序的版本等信息：

```
[root@master ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master.ilinux.io	Ready	master	25m	v1.12.1
node01.ilinux.io	Ready	<none>	1m	v1.12.1
node02.ilinux.io	Ready	<none>	1m	v1.12.1
node03.ilinux.io	Ready	<none>	32s	v1.12.1

到此为止，使用`kubeadm`构建Kubernetes集群已经完成。后续若有Node需要加入，其方式均可使用此节介绍的方式来进行。

A.2.7 获取集群状态信息

Kubernetes集群以及部署的插件提供了多种不同的服务，如此前部署过的API Server、`kube-dns`等。API客户端访问集群时需要事先知道API Server的通告地址，管理员可使用“`kubectl cluster-info`”命令了解到这些信息：

```
[root@master ~]# kubectl cluster-info
```

Kubernetes master is running at https://172.16.0.70:6443
CoreDNS is running at https://172.16.0.70:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

Kubernetes集群Server端和Client端的版本等信息可以使用“`kubectl version`”命令进行查看：

```
[root@master ~]# kubectl version --short=true
Client Version: v1.12.1
Server Version: v1.12.1
```

A.3 从集群中移除节点

运行过程中，若有节点需要从正常运行的集群中移除，则可使用如下步骤来进行。

1) 在Master上使用如下命令“排干”（迁移至集群中的其他节点）当前节点之上的Pod资源并移除Node节点：

```
~]# kubectl drain NODE_ID --delete-local-data --force --ignore-daemonsets
~]# kubectl delete node NODE_ID
```

2) 而后在要删除的Node上执行如下命令重置系统状态便可完成移除操作：

```
~]# kubeadm reset
```

A.4 重新生成用于节点加入集群的认证命令

如果忘记了记录Master主机的`kubeadm init`命令执行结果中用于让节点加入集群的`kubeadm join`命令及其认证信息，则需要分别通过`kubectl`获取认证令牌及验证CA公钥的哈希值。

“`kubeadm token list`”能够获取到集群上存在认证令牌，定位到其中DESCRIPTION字段中标识为由“`kubeadm init`”命令生成的行，其第一段TOKEN中的令牌即为认证令牌。而验证CA公钥的哈希值（`discovery-token-ca-cert-hash`）的获取命令则略微复杂，其完成格式如下所示：

```
~]# openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin  
-outform der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.* //'
```

而后，将上述两个命令生成的结果合成为如下格式的`kubeadm join`命令即可用于让Node加入集群中，其中的**TOKEN**可替换为上面第一个命令的生成结果，**HASH**可替换为第二个命令的生成结果：

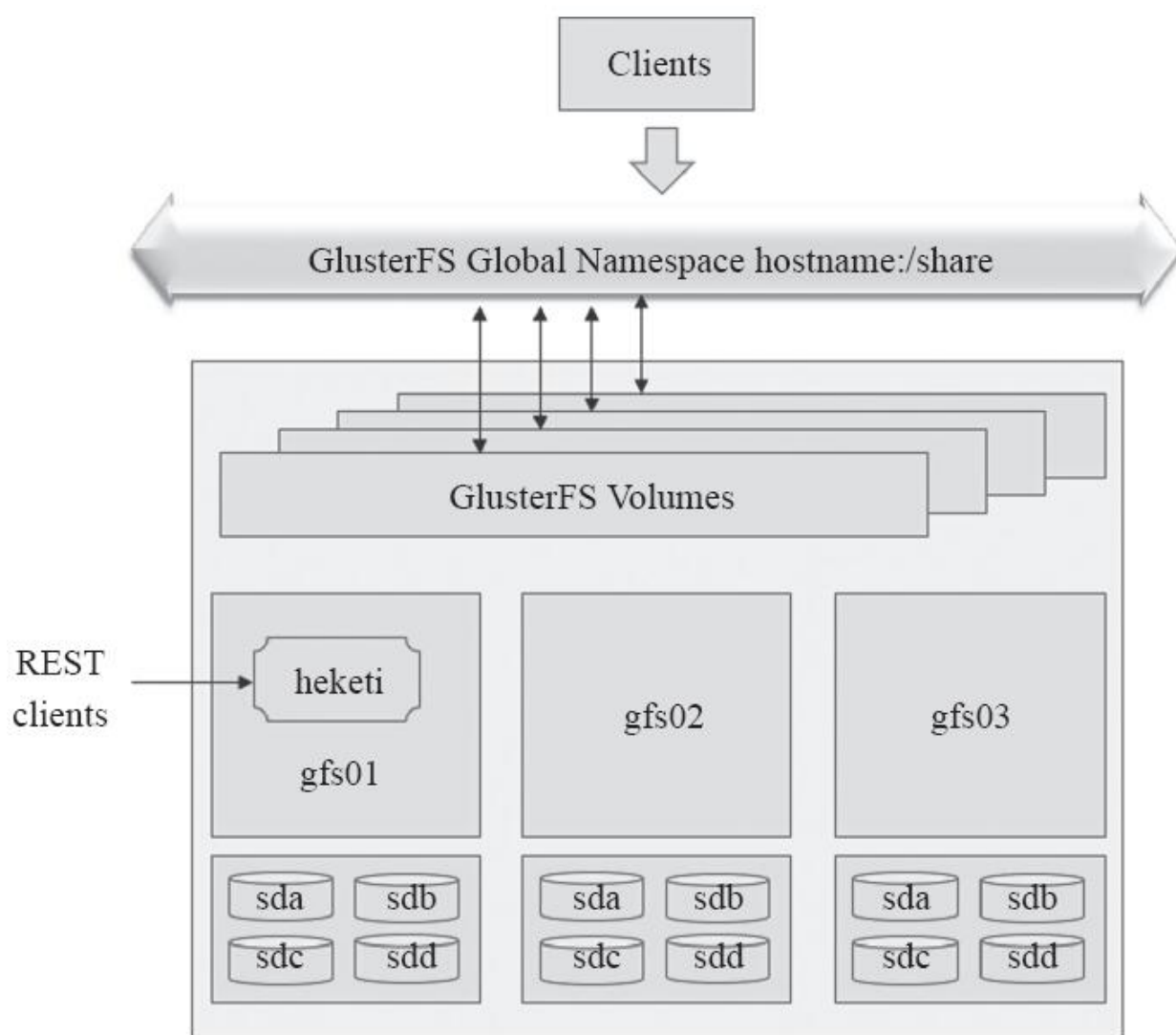
```
kubeadm join 172.16.0.70:6443 --token TOKEN --discover-token-ca-cert-hash HASH
```

最后，需要提醒读者注意的是，Kubernetes项目目前处于快速迭代时期，其版本号演进速度较快，因此，读者在测试时默认安装的程序版本与本书使用的极有可能存在着不同，甚至连部署步骤都可能会发生改变。建议读者参考作者的微信公众号iKubernetes以获取较新的部署文档，微信公众号二维码如下。



附录B 部署GlusterFS及Heketi

在实践Kubernetes的StatefulSet及各种需要持久存储数据的组件或功能时，通常会用到PV的动态供给功能，这就需要用到支持此类功能的存储系统。在各类支持PV动态供给功能的存储系统中，GlusterFS的设置较为简单，因此本书用到的各类动态存储功能将以此为例进行说明。本章将试图为读者提供一个设置以满足此功能的GlusterFS存储系统的简单说明文档，而不是对GlusterFS进行全面介绍。GlusterFS的架构如图B-1所示。



图B-1 GlusterFS架构

本示例中， gfs01.ilinux.io 、 gfs02.ilinux.io和gfs03.ilinux.io三个节点组成了GlusterFS存储集群（如图B-1所示），并将gfs01节点部署为heketi服务器。各节点上， sdb 、 sdc和sdd用于为GlusterFS提供存储空间。

B.1 部署GlusterFS集群

首先，分别在三个节点上安装glusterfs-server程序包，并启动glusterfsd服务，命令如下：

```
# yum install centos-release-gluster
# yum --enablerepo=centos-gluster*-test install glusterfs-server
# systemctl start glusterd.service
```

第二步，在任一节点上使用“glusterfs peer probe”命令“发现”其他节点，组建GlusterFS集群。命令格式为“peer probe{<HOSTNAME>|<IP-address>}”，例如，在gfs01上运行如下命令：

```
[root@gfs01 ~]# gluster peer probe gfs02.ilinux.io
[root@gfs01 ~]# gluster peer probe gfs03.ilinux.io
```

第三步，通过节点状态命令“gluster peer status”确认各节点已经加入同一个可信池中（trusted pool）：

```
[root@gfs01 ~]# gluster peer status
Number of Peers: 2

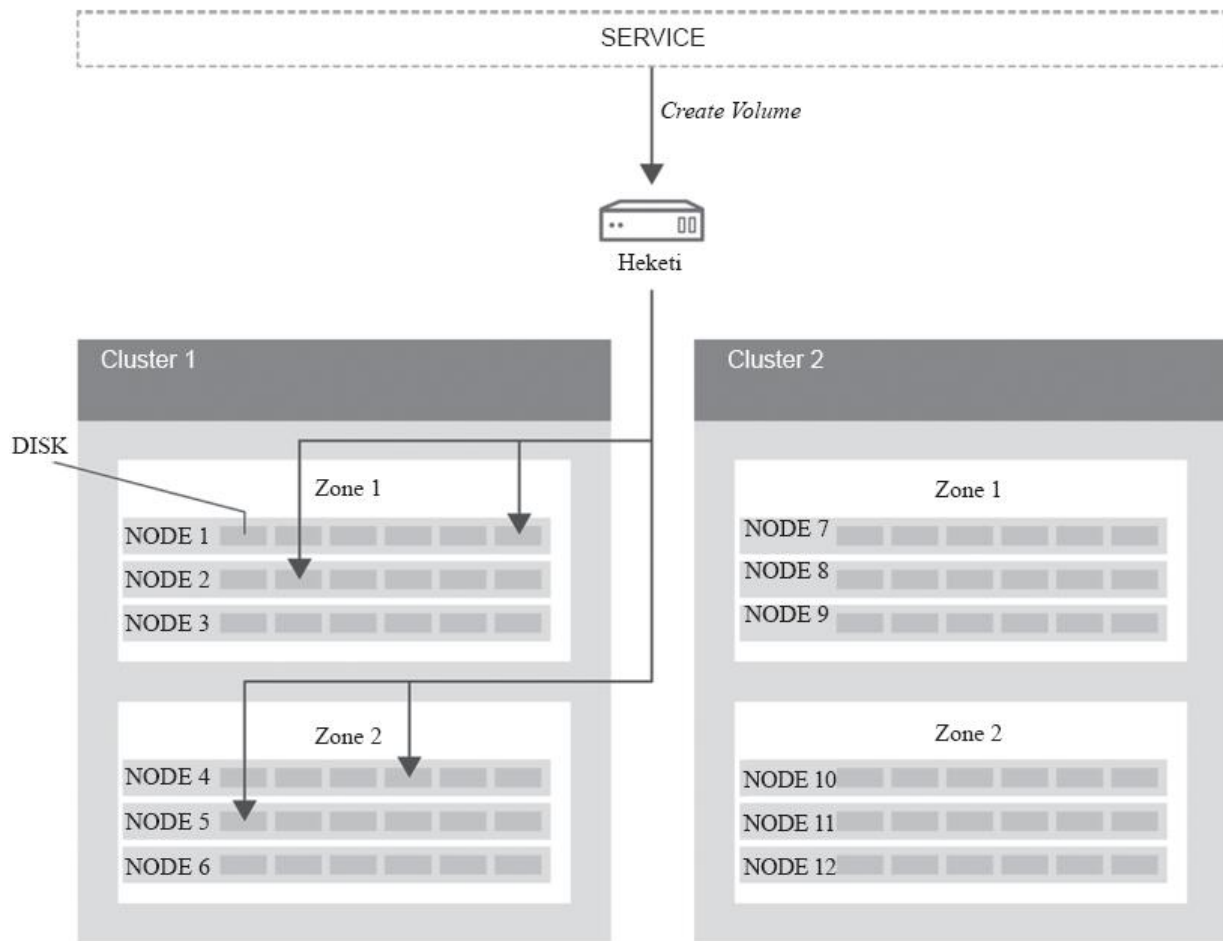
Hostname: 172.16.2.37
Uuid: e0f7e3cf-f856-4f95-a7d9-ad2e99cc455b
State: Peer in Cluster (Connected)
Other names:
gfs02.ilinux.io

Hostname: 172.16.2.38
Uuid: a1025d59-ac33-4265-a4d8-667b2d8a3a5c
State: Peer in Cluster (Connected)
Other names:
gfs03.ilinux.io
```

B.2 部署Heketi

Heketi为管理GlusterFS存储卷的生命周期提供了一个RESTful管理接口，借助于Heketi，像OpenStack Manila、Kubernetes和OpenShift这样的云服务可以动态调配Gluster存储卷。Heketi能够自动确定整个集群的brick位置，并确保将brick及其副本放置在不同的故障域中。另外，Heketi还支持任意数量的Gluster存储集群，并支持云服务提供网络文件存储。

有了Heketi，存储管理员不必再管理或配置brick、磁盘或可信存储池（trusted pool），Heketi服务将为管理员管理所有硬件，并使其能够按需分配存储。不过，在Heketi中注册的任何磁盘都必须以原始格式提供，而不能是创建过文件系统的磁盘分区。Heketi架构如图B-2所示。



图B-2 Heketi架构

本部署示例将gfs01.ikubernees.io节点用作Heketi服务器，因此以下所有步骤均在gfs01上执行。

B.2.1 安装并启动Heketi服务器

首先安装Heketi。Heketi程序可于epel仓库中获取，配置好相关的仓库后即可运行如下安装命令：

```
[root@gfs01~]#yum install heketi heketi-client
```

第二步，配置Heketi用户能够基于SSH密钥的认证方式连接至GlusterFS集群中的各节点，并拥有相应节点的管理权限：

```
# ssh-keygen -f /etc/heketi/heketi_key -t rsa -N ''
# chown heketi:heketi /etc/heketi/heketi_key*
# for host in gfs01 gfs02 gfs03; do \
    ssh-copy-id -i /etc/heketi/heketi_key.pub root@${host}.ilinux.io; done
```

第三步，设置Heketi的主配置文件/etc/heketi/heketi.json，定义服务监听的端口、认证及连接Gluster存储集群的方式。一个配置示例如下：

```
{
  "port": "8080",
  "use_auth": false,
  "jwt": {
    "admin": {
      "key": "admin Secret"
    },
    "user": {
      "key": "user Secret"
    }
  },
  "glusterfs": {
    "executor": "ssh",
    "sshexec": {
      "keyfile": "/etc/heketi/heketi_key",
      "user": "root",
      "port": "22",
      "fstab": "/etc/fstab"
    },
    "db": "/var/lib/heketi/heketi.db",
    "loglevel": "debug"
  }
}
```

若要启用连接Heketi的认证，则需要将“use_auth”参数的值设置为“true”，并在“jwt{ }”配置段中为各用户设定相应的密码，用户名和密

码都可以自定义。“glusterfs{ }”配置段用于指定接入Gluster存储集群的认证方式及认证信息。



提示 若启用了认证功能，则于Kubernetes集群中配置存储类时需要设置相应的认证信息。

第四步，启动Heketi服务：

```
# systemctl enable heketi
# systemctl start heketi
```

需要注意的是，将Gluster存储集群的功能托管于Heketi之后便不能够再于集群中使用命令管理存储卷，以免与Heketi数据库中存储的信息不一致。

第五步，向Heketi发起访问测试请求，无须认证时，使用curl命令即能完成测试：

```
# curl http://gfs01:8080/hello
Hello from Heketi
```

若Heketi启用了认证功能，则需要使用heketi-cli命令进行测试，命令格式如下：

```
heketi-cli --server http://<server:port> --user <user> --secret <secret> cluster
list
```

B.2.2 设置Heketi系统拓扑

拓扑信息用于让Heketi确认可使用的节点、磁盘和集群，管理员必须自行确定节点故障域和节点集群。故障域是赋予一组节点的整数值，这组节点共享相同的交换机、电源或其他任何会导致它们同时失效的组件。管理员必须确定哪些节点构成一个集群，Heketi使用这些信息来确保跨故障域中创建副本，从而提供数据的冗余能力。Heketi支持

多个Gluster存储集群，这为管理员提供了创建SSD、SAS、SATA或为用户提供特定服务质量的任何其他类型的群集选项。

命令行客户端**heketi-cli**通过加载预定义的集群拓扑，从而添加节点到集群中，以及将磁盘关联到节点上。要使用**heketi-cli**加载拓扑文件，可通过以下命令完成：

```
# export HEKETI_CLI_SERVER=http://<heketi_server:port>
# heketi-cli topology load --json=<topology_file>
```

一个适用于当前配置环境的示例配置如下所示（`/etc/heketi/topology_demo.json`），它将根据Gluster存储集群的实际环境把gfs01、gfs02和gfs03三个节点定义在同一个集群中，并指明各节点上可用于提供存储空间的磁盘设备：

```
{
  "clusters": [
    {
      "nodes": [
        {
          "node": {
            "hostnames": {
              "manage": [
                "172.16.2.36"
              ],
              "storage": [
                "172.16.2.36"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb",
            "/dev/sdc",
            "/dev/sdd"
          ]
        },
        {
          "node": {
            "hostnames": {
              "manage": [
                "172.16.2.37"
              ],
              "storage": [
                "172.16.2.37"
              ]
            },
            "zone": 1
          },
          "devices": [
            "/dev/sdb",
```

```

        "/dev/sdc",
        "/dev/sdd"
    ],
    },
    {
        "node": {
            "hostnames": {
                "manage": [
                    "172.16.2.38"
                ],
                "storage": [
                    "172.16.2.38"
                ]
            },
            "zone": 1
        },
        "devices": [
            "/dev/sdb",
            "/dev/sdc",
            "/dev/sdd"
        ]
    }
]
}

```

而后运行如下命令加载拓扑信息，从而完成集群配置。此命令会生成一个集群，并为其添加的各节点生成随机ID号：

```

# export HEKETI_CLI_SERVER=http://gfs01.ilinux.io:8080
# heketi-cli topology load --json=topology_demo.json

```

而后运行如下命令查看集群的状态信息：

```

# heketi-cli cluster info ba7657cf60195432d8c0b4e3c2c94d59
Cluster id: ba7657cf60195432d8c0b4e3c2c94d59
Nodes:
2a73c6123e4a05ceb050bc5c433c6a07
613754fe6267e3d7f1a731ce2e0ab94a
8b4e57a8ce4e5640b6cc845915769f8d
Volumes:

```

“**heketi-cli volume create --size=<size in Gb> [options]**”能够创建存储卷，例如，下面的命令测试即用于创建一个存储卷：

```

# heketi-cli volume create --size=20
Name: vol_43b859b65d4ad8e0de47065b2aca7e41
Size: 20

```

```
Volume Id: 43b859b65d4ad8e0de47065b2aca7e41
Cluster Id: ba7657cf60195432d8c0b4e3c2c94d59
Mount: 172.16.2.36:vol_43b859b65d4ad8e0de47065b2aca7e41
Mount Options: backup-volfile-servers=172.16.2.37,172.16.2.38
Durability Type: replicate
Distributed+Replica: 3
```

而后在要使用的远程存储卷的节点上安装GlusterFS和glusterfs-fuse程序包，提供GlusterFS客户端驱动及对GlusterFS文件系统的运行，并基于GlusterFS文件系统类型挂载使用确认无误后即可删除测试卷。删除Heketi卷的命令为“`heketi-cli volume delete<vol_id>`”，如删除前面创建的存储卷，可使用以下命令：

```
# heketi-cli volume delete 43b859b65d4ad8e0de47065b2aca7e41
Volume 43b859b65d4ad8e0de47065b2aca7e41 deleted
```

至此为止，一个支持动态存储卷配置的GlusterFS存储集群即设置完成，用户既可于Kubernetes中通过PVC请求使用某事先创建完成的GlusterFS存储卷，也可把Heketi配置为存储类，而后提供PV的动态供给功能。